An Integrated Instrumentation Environment

Technical Report 86-028

1986

Richard R. Morrill

The University of North Carolina at Chapel Hill
Department of Computer Science
New West Hall 035 A
Chapel Hill, N.C. 27514

AN INTEGRATED INSTRUMENTATION ENVIRONMENT

by

Richard Raymond Morrill

A Thesis submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

1986

Approved by:

[illegible]
Adviser

[illegible]
Reader

[illegible]
Reader

Acknowledgements

The SoftLab team provided invaluable support and aid during the entire thesis project. In particular, Rick Fisher helped with the design and implementation of the IIE features that depended on his family of operating systems. Rick Snodgrass as both the head of SoftLab and the advisor for this thesis was always available for help and guidance. His insight in the midst of some difficult design decisions kept things on track. Dr. M. Satyanarayanan provided assistance in getting his hardware simulator ported and adapted for our use in SoftLab.

Special thanks to Janice, Geoffrey, and Paul. It won't happen again.

TABLE OF CONTENTS

1. Introduction

The SoftLab project strives to treat both education and research in computer science as in natural sciences. Natural sciences and computer science differ primarily in the experimentation area. Physics, for example, relies heavily on laboratory use for both education and research. Over the years physicists have developed many special tools to support these experiments. There are now some laboratory courses in place for the hardware area of computer science. The SoftLab group set out to complement these by developing support for experiments in the software areas of computer science.

Physicists have been developing experiment support tools for hundreds of years. It is not realistic to think that a small research group can develop tools to support all of the different areas of software research and education. We must start by providing support for experiments in a few particular areas, doing so with generality in mind so that we can easily extend this support, a little at a time, to cover a wider and wider range of experiment types.

The selected areas for initial support are operating systems and compiler design. These choices led to the following major tasks:

1. the design and implementation of a modularized compiler.

2. the design and implementation of a family of modularized operating systems.

3. the design and implementation of an environment to support compiler and operating systems experiments.

As part of the first task, members of the SoftLab team designed and implemented the Interface Description Language(IDL) [18]. This language is a general purpose interface description tool that is especially suited to aid in building compilers with multiple passes. As part of task two, other SoftLab members are nearing completion on the design and implementation of two modular operating systems [7]. This task required additional compiler support to assure that the operating systems, written in Modula-2, could run on a hardware simulator. Task three requires a number of support programs, e.g., a hardware simulator, to provide necessary experiment components. In addition, this task requires the design and implementation of a set of support tools that bring all of the experiment components together and perform a simply specified experiment.

Our involvement in the above tasks was in three main areas: for task two we assisted in the design of the Multibatch operating system; as part of task three we were involved in modifying an existing hardware simulator [16] and an M-code, Modula-2 intermediate code, interpreter to support the SoftLab operating systems; and we designed and implemented the support tools for aiding experiment specification, synthesis, execution and evaluation.

For this thesis a set of tools, which comprises an Integrated Instrumentation Environment (IIE) for architectures and operating systems, will be designed and implemented. This IIE will assist users with a wide range of experiments relating to both hardware and software configuration. As part of this effort a language will be designed for the purpose of specifying valid experiments.

Chapter 2 describes the problems involved with experiments in general and goes on to show the particular problems related to software system experimentation.

Chapter 3 presents the SoftLab approach to software system experimentation and a brief description of the present SoftLab tool set.

Chapter 4 contains the User Manual for the SoftLab Integrated Instrumentation Environment.

The design and implementation of SoftLab's Integrated Instrumentation Environment is the subject of chapters 5 and 6. We discuss design and implementation decisions as well as the value of capitalizing on the UNIX toolset to do rapid prototyping.

In the conclusion, we argue both the generality and utility of SoftLab's IIE and present some of the work to be done to extend its present capabilities and enhance its structure.

2. Experiments

Scientific experimentation is generally a very complex and lengthy endeavor. If an experimenter is to draw reasonable conclusions at the end of an experiment he must know the state of the experiment initially, while in progress, and at termination. For example, a particular experiment in the field of Chemistry may require temperature, humidity, and barometric pressure readings while in progress. A chemist may wish to rerun such an experiment with the same ambient temperature and humidity but with a different barometric pressure; then, at some future date he may wish to repeat the original experiment. Experimental chemists have designed special tools over the years to aid them in this kind of control process. These special tools encourage many experiments, and series of experiments, that would otherwise be too difficult.

2.1 Software Systems Experiments

The problems involved in experimentation in Computer Science are similar to those of other sciences like Chemistry. They are the problems related to repeatability, control, and modification of experimental parameters. All of these issues are important in the area of operating systems experimentation. An operating systems experiment generally consists of four main parts: a set of connected hardware devices (e.g., line printer, cpu, card reader); one or more user programs (the *workload*); an operating system; and a mechanism or set of mechanisms to collect data concerning the state of the experiment. Each aspect carries with it a set of problems which complicate experimentation. The particular problems related to each of these four areas, as well as some others, are mentioned below. For the remainder of this section 'experiment' refers to 'operating systems experiment'.

- In most cases an experiment requires the dedicated use of the hardware devices while in progress. If particular modifications to the hardware are required as part of a set of experiments, these devices cannot generally be shared. Hardware may have to be acquired if it is necessary to test the operation of an operating system over a range of equipment —which makes many operating systems experiments too costly to perform.

- It is not easy to specify workloads for experiments. A program, or set of programs, must be carefully written so that it possesses the characteristics required for the correct evaluation of a particular experiment. If any of the workload characteristics must vary over a set of experimental runs, different workloads with the correct variations in characteristics must be composed.

- Operating systems that are currently in use are generally very large and complex. Even the simpler operating systems such as UNIX are composed of more than 10,000 lines of code. Since efficiency is of paramount importance if an operating system is to be useful, the code is often very dense and hard to understand.

- The evaluation of most operating system experiments requires the collection of data during the experimental run. The data collection mechanism must have no substantive effect on the experiment. This mechanism may well involve modifying existing hardware to provide certain information. The data collected during an average experiment may also be voluminous. For example, information concerning the number of memory references during the execution of a large program may be required. The data collection mechanism should then contain some means to filter or process this data as it is being collected. If the mechanism does not provide such means then the processing of the data after the experiment will be a major task in itself.

- Experiments which involve classroom assignments can lead to special problems. Many students may have to carry out the same experiment in roughly the same time frame. Protection issues become difficult to handle when many students are sharing a large number of files.

It is evident that the problems encountered in experimentation render it too costly in many of the situations where it would be useful. Very few universities have the resources that would allow students the kind of opportunities to experiment with operating system design and evaluation that many educators and the SoftLab group feel they should have. We may be able to solve this difficulty by developing tools for the express purpose of supporting experiments just as others in the sciences have done in their respective fields. One advantage that we have is that software systems experiments can be controlled by the same mechanisms (i.e., software) that are being controlled. This allows us a unique opportunity to study the properties of this interaction.

2.2 Previous Work in Software Systems Experiments

At this time there has been no project or proposed project that fully addresses the issues and associated problems with software system experimentation laid out in the previous section. There have been proposals that address laboratories and experiments in regard to specific topics, e.g., operating system or compiler design[1, 2, 11, 12, 19]. Only two proposals, however, address experimentation in more than one area of software system design.

One system, SL230[3, 4], which has been proposed at Carnegie Mellon University, would allow for experimentation with regard to operating system design. This system provides a large collection of component modules and a means to link them together via a message passing mechanism. A kernel also provides low level synchronization of the modules. The modules are written in assembly language to provide an efficient implementation. A user of the system can carry out experiments related to modular operating system design easily and efficiently if he restricts himself to modularized operating systems, based on a message passing paradigm, that can be synthesized from existing component modules. Since the modules are implemented in assembly language, implementing new modules or modify existing ones significantly complicates the experimentation process. These restrictions are clearly a problem if the intended use of the system is for sophisticated experimentation or for flexible pedagogical purposes. Another drawback of the system is its specific ties with operating systems. It does not seem to be the basis of a general purpose software system experimentation tool.

Halstead's system[10] proposes a laboratory for both operating system and compiler implementation. The system includes a modularized compiler for the Pilot language, and a modularized operating system written in Pilot. Experiments relating to the design and construction of operating systems or compilers are performed by modifying one or more of the constituent modules. This system is intended as a pedagogical tool only. The simplicity of the included language, Pilot, and the requirement of a bare machine allow little flexibility in the use of the system. Its use in the pedagogical environment is also limited by the requirement that each student implement most of the operating system and compiler with only basic tools (i.e., a text editor and a compiler).

3. SoftLab

The goals of the SoftLab project are to address and to solve as far as possible the problems associated with designing and carrying out software systems experiments. Attaining this goal requires a rich set of experiment components (e.g., operating systems, hardware simulators, etc.) in an environment which supports experiment design, execution, and control. This project involves designing and implementing a set of tools that comprises an Integrated Instrumentation Environment(IIE)[17]. An IIE supports software system experimentation in the way that a programming environment supports software system development. Programming environments provide programmers with the tools to support the design, implementation and maintenance of software systems. Some PE's provide a collection of tools from which the user can select [5]; others support the programmer through a particular phase of the software system development life cycle [6]; and still others provide support across the entire life of a program [8]. The IIE provides a rich set of tools when viewed from the point of view of the experiment designer. She can select an operating system, a compiler and a particular workload as components of an experiment. The experiment performer has the tools to take the experiment specification and then, step-by-step, take it to a working experiment. Additional components of the environment allow the experiment to be modified and run repeatedly without any danger of unspecified inconsistencies entering into the process.

An Integrated Instrumentation Environment will allow the various tools (e.g., machine simulators, families of operating systems, etc.) to function together and assist in performing a wide rage of software systems experiments. The IIE solves the problems of repeatability, specification, and control that are inherent in the experimentation process. It will also provide the interface to data collection and analysis tools, device simulators, and modularized software components (e.g., compiler passes) that are required for sophisticated experiments.

4. The IIE Users Manual

4.1 Introduction

An *Integrated Instrumentation Environment* (IIE) supports software system experimentation in the same ways that programming environments support software system development. Programming environments provide programmers with the tools to support the design, implementation and maintenance of software systems. Some PE's provide a collection of tools from which the user can select [5]; others support the programmer through a particular phase of the software system development life cycle [6]; and still others provide support across the entire life of a program [8]. The designer can select an operating system, a compiler and a particular workload as components of an experiment. The experiment performer has the tools to take the experiment specification and then, step-by-step, take it to a working experiment. Additional components of the environment allow the experiment to be modified and run again and again without any danger of unspecified inconsistencies entering into the process.

The IIE described in this manual is an integral part of the SoftLab environment. It provides the means to describe and carry out a wide range of software system experiments. We assume that the user is already familiar with the other SoftLab components that are a part of the planned experiments. In particular, we assume that the user is familiar with the SoftLab family of operating systems [7], the SoftLab hardware configuration simulator [16], the Modula-2 programming language [9, 22] and the C programming language [13].

Section 4.2 contains a brief description of the major runtime components of the IIE. Section 4.3 describes the experiment schema that specifies how these components should interact in a particular experiment. In Section 4.4 the user learns how to use the experiment preparation program "SchemaPrep." Section 4.5 illustrates the execution of an experiment and describes how to utilize pieces of an existing experiment in a related experiment. This can often save a lot of time and system resources. Section 4.6 presents a description of an example experiment from start to finish. The last chapter describes the error handling facilities that are part of the IIE.

4.2 Major IIE Runtime Components

The IIE incorporates many sophisticated components. The major runtime components of an experiment are the operating system, the hardware simulator and the stimulus, or workload. We present the user choices for each of these components below.

The Operating System

The SoftLab operating system family is a set of modularized operating systems. We call this set a family because of the many similarities among the various operating systems. There is a clear progression in complexity within the set. Two of the family members are currently being implemented, the *Unibatch* operating system and the *Multibatch* operating system. The former supports one CPU, one primary memory unit, a card reader and a line printer. It handles one job at a time. Jobs are entered as card decks and the output of the job is printed on the line printer. The Unibatch operating system is useful as a model of the most primitive type of operating system; even though it is very simple it illustrates the basic structure of an operating system. A simple experiment with this system might involve adding double buffering to the card reader driver.

The latter O/S supports one basic CPU, one primary memory unit, a disk drive, a card reader and a line printer. More than one job may reside in main memory at a given time. The jobs come into the system as separate card decks. The results of all of the jobs appear as hardcopy on the line printer. The Multibatch operating system represents the next step in complexity. The addition of multiple batch job processing capabilities carries with it an impressive jump in code size. The three level scheduling mechanism, the memory management module and the interrupt handler provide ample opportunity to investigate real operating system design decisions. These two operating systems are implemented in Modula-2.

The Hardware Simulator

Dr. Satyanarayanan, at Carnegie Mellon University, designed and implemented a flexible hardware simulator to aid in the design and investigation of sophisticated network file servers — this tool was adapted for SoftLab. Each component of the hardware is specified in a *device module* file. This file contains entry and exit points for both control and data, procedures that embody the component simulation algorithms and special simulation directives. One of these device modules is constructed for each of the components of the hardware that we wish to simulate. A *hierarchy description* file contains the component interconnection specifications and component initialization directives. A completed simulator consists of one hierarchy description file and any number of device module files. The implementation language for both file types is classc, a modified version of the C language with classes [20, 21]. Two simulators constructed with this tool are presently in use: the *Umachine*, which supports the Unibatch operating system, and the *Mmachine*, which supports the Multibatch operating system. The Umachine contains one CPU device module, a main memory device module, a card reader device module and a line printer device module. The Mmachine contains one CPU device module, a main memory device module, a card reader device module, a disk device module and a line printer device module. The main memory device module, the card reader device module and the line printer device module are identical in the two machines. Each CPU device module contains a slightly different M-code interpreter. M-code is the intermediate language for Modula-2; p-code is the analogous language for Pascal [15].

The Stimulus

The SoftLab stimulus facility consists of a number of libraries of workload component routines. Programs are written that reference these routines; the appropriate library is linked at compile time. This facility reduces the time and difficulty involved in defining and implementing a stimulus with specific properties. The libraries differ in the execution time of the routines and in target type. Two libraries, both intended for single processor machines, may contain corresponding routines that, although similar in function, differ by a factor of ten in running time for example. Other libraries may have a multi-processor machine as the intended target. Routines that differ in running time only have the same name by convention. The user selects workload duration at compile time by linking with the desired library.

Other IIE components

The operating system family, the hardware simulator creation tool and the stimulus facility are integral parts of the SoftLab IIE. They are also tools in their own right. Other IIE components are more tightly bound and do not have a useful separate existence. We will discuss their features in the context of the experiment specification facility in the following section.

4.3 Experiment Specification

The components of the IIE's runtime environment are powerful but complex. An additional facility is necessary to describe the interconnections of the components so that the user can easily specify and evaluate an experiment. The *Express* (Experiment Requirements Specification Schema) language provides this facility. The user writes a description of the experiment in this language. We call this description the *experiment schema*, hereafter known as the schema; we call the file that contains this description the *schema file*. Examples from a valid schema file `example.sch` are presented; the example file is given in the Appendix.

A schema file contains six major parts. Each of the parts specifies part of the runtime environment of the IIE. The six sections are:

Initialization Section

The body of this section provides the information necessary to select various initialization options of IIE runtime components. Possible initialization options are the memory size for a particular hardware configuration, the number of processors for a multiprocessor hardware specification and the per process stacksize limit for an operating system supporting multiple users. The initialization section is not supported in the IIE implementation.

Initialization Section from the file `example.sch`:

```
InitSection
            EndInit
```

Stimulus Section

The stimulus, or workload, exercises the runtime environment and is necessary for any real experiment to take place. It is possible to specify an experiment that has an empty workload, but outside of testing the initialization features of the hardware configuration and operating system it is not useful to do this.

In the stimulus section the user may specify a SoftLab stimulus library and a stimulus program. The stimulus library contains routines that the stimulus program imports. The library routines each embody a real user workload feature such as I/O-bound code, compute-bound code or code exhibiting little locality. The user stimulus program must be a valid Modula-2 program. The program may import SoftLab stimulus routines only from the specified library.

Stimulus Section from the file `example.sch`:

```
StimulusSection
               Library:  lowref
               File:  /unc3/unc/drm/sl/src/Stimulus/main.mod
               EndStimulus
```

The stimulus library `lowref` is selected along with user written driver `main.mod`.

Hardware Configuration Section

The user must choose one of SoftLab's hardware simulators. He must also specify the configuration of these components by naming a hardware configuration file. These two selections taken together specify a valid hardware simulator, called the **base machine**. Additional schema file entries for the hardware configuration section specify modifications to the base machine. These modifications consist of pairs of device module file names. The first file name specifies one of the device modules that is part of the base machine; it is assumed to be in the base machine directory. The second file name specifies a substitute device module and may be an arbitrary path name. The resulting hardware simulator consists of the base machine with the specified substitute device modules.

Hardware Configuration Section from the file example.sch:

```
HWConfigSection
              Machine:  Bmachine
              HD: Bmachine.hd
              DMSubs:
                            Bmainmem.dm submem.dm
                            Bcpuint.dm subcpu.dm
              EndSubs
              EndHW
```

The hardware simulator Bmachine is selected along with its primary hierarchy description file Bmachine.hd. Other hierarchy description files for the hardware simulator Bmachine are also possible, specifying different variations of the same machine. The device module files Bmain.mem and Bcpuint.dm are replaced with the files submem.dm and subcpu.dm respectively. The replacement files are in the schema file directory since no path components are specified.

Operating System Configuration Section

The operating system configuration section is similar to the hardware configuration section. It starts with the choice of an operating system from the SoftLab family of operating systems. The next selection is the main module of the Modula-2 program that implements the chosen operating system. This main module is similar in function to the hierarchy description file for the hardware configuration section. The choice of different main modules specifies different variations of the same basic operating system. In each case the final program is built from selections from the same pool of modules. The user may now choose to substitute modules in the operating system. Since Modula-2 programs incorporate both definition modules and implementation modules, the user is able to substitute for either module type. The resulting operating system consists of the basic operating system selected with the specified modules incorporated.

Operating System Configuration Section from the file example.sch:

```
OSConfigSection
              OS: unibatch
              Main:  UniBatch.mod
              DefSubs:
                            Loader.def /unc/drm/ms_work/Imp/test/subloader.def
              EndDefSubs
              ImpSubs:
```

```
                        Loader.mod /unc/drm/ms_work/Imp/test/subloader.mod
        EndImpSubs
        EndOS
```

The **unibatch** operating system and its main module **UniBatch.mod** are selected. The definition module **Loader.def** is replaced. Notice that here the full path name of the substituted file is given. The implementation module **Loader.mod** is also replaced.

Sensor Section

Each of the components of the SoftLab hardware simulators, operating systems and stimulus libraries contain sensors. These sensors are embedded code fragments that supply information from the runtime system to the experiment manager. In addition, the user may define new sensors. These additional sensors could be found in the user stimulus program, substitute device modules or substitute operating system modules. Each sensor has a unique name. The experiment manager selects the information from the sensors that are of interest to the experimenter. The user may name the sensors of interest directly in this section of the schema or indirectly by naming a file that includes a list of the sensors after the switch **-f**. Multiple sensor files are acceptable but they must each be proceeded by **-f**. The documentation for specific SoftLab components contains lists of the sensors for that component. The SoftLab monitor document contains the description of a sensor code fragment.

Sensor Section from the file **example.sch**:

```
SensorSection
        SensorList:   1 5 8
                      -f senslist
                      45
        EndSensorList
        EndSensor
```

The sensors 1, 5, 8 and 45 are selected explicitly in this section. The sensors listed in the file **senslist** in the schema file directory are also selected. An arbitrary pathname could have been used for the sensor file.

Directives Section

This section specifies the number of runs of the simulation system that are to take place in the experiment as well as the variations for each run. The only per run variation that we allow at this time is the choice of stimulus. Thus, for example, the user may direct the experiment manager to execute the simulation system three times with a different stimulus for each run via three separate run-tuples. Each stimulus is a selection from the stimuli built in the stimulus section. The five elements in each run-tuple are: first, the base machine; second, the executable operating system file name; third, the executable stimulus file name; fourth, the constructed sensor list file name; fifth, the output file name.

Directives Section from the file example.sch:

```
DirectivesSection
            RunList:      ("Bmachine","UniBatch.out","main.out","sensors","out1")
                          ("Bmachine","UniBatch.out","main.out","sensors","out2")
            EndRunList
            EndDirectives
```

Two runs of the simulator are specified in this section. At present the only possible variation in these run-tuples is different names for the run output files. In this case the file out1 will contain the output from the first run and the file out2 will contain that of the second. In the future any of the run components listed in the run-tuple, the simulator, the operating system, the stimulus, the sensors and the output file may change from run to run. The corresponding sections of the schema file will also change to allow more than one run component, e.g., two operating systems, to be specified.

Summary

The schema file contains a complete specification of a SoftLab software experiment running under the IIE. Modifications to a schema file represent new experiments. These modified schemas, however, make the differences in each experiment clear, and provide an easy means for the user to specify a series of experiments in a controlled manner with small variations at each step. The next chapter contains a description of the tool that interprets a schema file and prepares the components of the experiment for execution. The Appendix contains the description of a valid schema file.

4.4 Experiment Preparation

A schema file describes an experiment by specifying each of the major components of the IIE runtime system and also by issuing directives of which the experiment manager will take note during experiment execution. The program `SchemaPrep` (see `SchemaPrep` man page in Appendix C) takes this schema file as input and prepares the experiment components for execution. In addition to naming the schema file the user may also set command line switches. Setting a switch causes `SchemaPrep` to process the associated section of the schema. If no switch is set the program processes all sections of the schema. Sections of the schema that `SchemaPrep` ignores, due to switch settings, must still contain the correct delimiting keywords. The experimenter invokes `SchemaPrep` as follows:

`SchemaPrep [-dhimow]` *schema file*

The switches have the following meanings:

`-d` Interpret the directives section

`-h` Interpret the hardware configuration section

`-i` Interpret the initialization section

`-m` Interpret the sensor section (for monitoring)

`-o` Interpret the operating system section

`-w` Interpret the stimulus section (the workload)

The user must also include the directory `/usr/softlab/bin` in the environment variable `PATH` string. See the Unix manual entry on `csh` to find out how to do this.

The `SchemaPrep` program reports its progress on interpreting the schema to both standard output and the file `SchemaPrep.log` in the current directory. When the program encounters errors in a schema file section it reports the difficulty and attempts to parse subsequent sections of the schema. The user may call `SchemaPrep` with the appropriate switches set to parse sections that were incorrect on a previous execution. If the file `SchemaPrep.log` is present `SchemaPrep` appends any new messages to the end of the file. This provides a complete account of the preparatory phase of the experiment.

If the user first executes SchemaPrep in a directory containing only the schema file, the result of a successful parse of the schema is the creation of a number of new sub-directories and files. For example, if we started in a directory containing the following files:

```
example.sch    senslist      subcpu.dm      submem.dm
```

and invoked SchemaPrep on the example described in Chapter 3, a successful parse of the schema file would leave us with the following additional files

```
SchemaPrep.logmakefile      senslist      subcpu.dm      submem.dm
```

and subdirectories

```
.exp_mgr:
.hwconfig:
.osconfig:
.stimulus:
```

each containing a `makefile` and other files.

Executing the `makefile` in the schema file directory will cause each of the subdirectory `makefiles` to execute in turn. If errors cause any of the `makefiles` to stop before making the associated component, the user can correct the problem and then run SchemaPrep on the problem section. The higher level `makefile` will remain, and execution of the `makefile` will continue with the problem subdirectory. When the upper level makefile completes its execution it produces an executable file exp_mgr in the schema file directory. This program executes the experiment as specified in the schema. The appendix shows this process in more detail.

4.5 Experiment Execution

The user executes the experiment specified in the schema file by executing the program exp_mgr (see exp_mgr man page Appendix C) in the schema file home directory. The experiment will then run until all of the requested runs complete. The results of the experiment are written to the files named in the run-tuples in the associated schema. In addition to the results files, exp_mgr produces a file containing a log of the experiment in the same directory. This file is named `exp_mgr.log`. The log file contains information on the experiment specifications, the time the experiment was run and any warning or error messages produced as a result of the execution. The logging mechanism is not currently implemented. The user can obtain the same information from the other log file, SchemaPrep.log, and from what is printed to standard output during the experiment run. The results file contains the formatted output of the experiment with the information requested in the sensor section of the schema. In the example execution (see Appendix B) two runs are carried out with the names of the output files the only difference.

The files out1 and out2 contain the following information:

- the sensor id;
- the first time the sensor was reached, in simulation time units;
- the last time the sensor was reached, in simulation time units;
- the average time between sensor events;
- and the number of times the sensor was reached.

Any execution of an exp_mgr program derived from the same schema will produce the exact same result and log files. The only obvious exception is the experiment time information in the log file.

The user may modify a schema file after executing an experiment. SchemaPrep switches are set to select the modified sections of the schema for processing. The SchemaPrep.log file will contain the new experiment preparation log information in addition to the previous contents. The user should delete this file before running SchemaPrep if she wants it to contain just the new information. The new exp_mgr program and the files resulting from its execution will overwrite the previous files. A user wishing to run more than one experiment using modified versions of the same schema file in the same directory should copy each set of log and result files before the next experiment. A full example of an experiment from the preparatory phase to completion is in Appendix B of this manual.

4.6 Error Handling

SchemaPrep and exp_mgr can produce three types of errors. The steps for the user to take are evident from the initial message: she should consult the appropriate language manuals, SoftLab manuals, or local experts for help.

Configuration Errors

These errors are the result of naming files in the schema that do not exist or do not have the appropriate permissions. This includes naming non-existent SoftLab components. Configuration errors produce clear messages that pinpoint the difficulty.

Syntax Errors

These errors result from incorrect syntax in user written experiment components such as substitute device modules or a stimulus program, and from syntactic errors in the schema file. Syntax errors in the schema file may lead to additional error messages associated with subsequent schema sections that are correct. The user should correct the schema file at the place where the first error occurred and then run `SchemaPrep` again before attempting any other modifications.

Runtime Errors

These are the result of a variety of errors in the user written Modula-2 routines or C language routines.

5. The Design

This chapter presents the overall design of the IIE introduced in Chapter 4. We include herein materials on the design of the tools for specifying and managing an experiment, the modification and incorporation of the main components presented in Chapter 3, and the design of the stimulus and monitoring facilities. We assume the reader is familiar with the definitions and terminology presented in the previous chapter.

5.1 Experiment Support

The IIE requires tools to support experiment specification and management, in addition to a set of experiment components (e.g., hardware simulators). The framework for the design of these support tools (including those for the contents of the schema file) derives from the work of Segall, et al, at Carnegie Mellon University on the design of an IIE [17].

The experiment process seems to fall naturally into five main steps: 1) the specification of the experiment; 2) the preparation and testing of the specified experiment components; 3) the construction of the experiment components; 4) the execution of the experiment; and 5) the presentation of the experiment results. We now discuss each step in more detail.

- The first step involves analyzing the requirements of an experiment and then writing the appropriate specification. The specification of the experiment, termed the *experiment schema*, is written in the Express language.
- In the second step, component specifications are checked, and necessary preparation for component construction (e.g., moving files to a particular directory) takes place. The `SchemaPrep` program carries out the duties for step two.
- During the third step, the actual construction of each component occurs. The UNIX `make` utility provides the function for this step, including the conditional processing necessary to avoid repeating the checking and preparation of all components when only one requires it. Handling the experiment preparation phase in this way supports a batch oriented approach to each stage of the experiment process, and is vital if fifty students will carry out the same experiment over the period of a week or two.
- Managing an experiment requires experiment initialization, execution and run-time control. These tasks are well suited to an implementation with multiple processes, each with separate

duties. The incorporation of inter-process communication into the design is beyond the scope of this thesis and must be left for future work. For this reason, the design for the experiment manager program **exp_mgr** should admit a straightforward uni-process implementation.

- The data collected during an experiment execution may require significant post-processing to be comprehensible, and depend on a sophisticated data storage mechanism along with a powerful data processing tool. The design and implementation of such tools is beyond the scope of this thesis. The analysis and presentation of data in this design is simple enough to require a minimum of implementation effort, though still support real experiments.

5.2 Experiment Specification

Schema Design Considerations

The design of the schema was directed by three primary considerations, namely:

- that it support the specification of a wide range of O/S experiments;
- that it allow easy extension to encompass new experiment components; and
- that it be readable and self-explanatory.

The Schema Contents

The specification of an O/S experiment starts with the selection of the experiment components. The operating system, hardware simulator, and stimulus are the primary elements and require attention in the schema. The schema must also provide the means to direct the monitor to collect the necessary data during an experiment run. Real experiments may consist of related sets of runs; therefore, the schema should enable a user to specify multiple runs of an experiment, where each run may contain different selected components. Particular components of an experiment may include instantiation parameters, allowing, for example, the selection of the number of processors in a multi-processor configuration or the per-process stack size limit in a multi-user operating system.

Separate sections in an experiment schema distiguish experiment configuration, data collection, and management specifications. Breaking the schema into distinct sections provides easy extension; we can add a new section without interfering with the contents of the others.

Initialization Section

Within this section we are able to set instantiation parameters. The design does not provide more than the existence of this section at this time. Design decisions of greater detail should await additional component specifications. It is too difficult at this time to gauge the range of possible instantiation directives.

Stimulus Section

The selection of a SoftLab stimulus library and a particular file that uses routines from the library constitutes the specification of a give workload. The stimulus library designers can provide as extensive a set of individual routines as they see fit. It seems more appropriate to keep this section simple and leave the workload complexity issues to stimulus writers. The design can easily encompass modifications to allow the specification of more than one workload through minor changes to the Express language and SchemaPrep.

Hardware Configuration Section

The SoftLab approach for O/S experiments includes the notion of a set of hardware simulators from which the experimenter selects. Since there may exist more than one hierarchy description file for a particular simulator, this section includes the selection of this file in addition to the particular machine. A list of device module substitutions in this section allows the selection of a machine variant. This section along with possible instantiation parameter settings should provide the necessary range of hardware selection.

Operating System Configuration Section

The content of this section follows from the structure and intended use of the SoftLab family of operating systems. We allow the selection of an operating system by name, followed by the selection of a main module, since more than one may exist. Modula-2 programs contain both definition and implementation modules; this calls for a section for each type of module substitution. Additional selection specifications do not seem necessary.

Sensor Section

We include sensor files in the sensor list for experiments requiring the enabling of a common subset of sensors. A simple extension will allow for the construction of more than one sensor list. Future designs may also include the ability to enable and disable sensors while a run is in progress. The design of such a feature is beyond the scope of this thesis, since it involves complex interactions among the components during a run.

Directives Section

This section was the most difficult to design. The choice to compose a list of run-tuples specifying the experiment components was due in part to ease of implementation. This design does not require any re-building of components or inter-process communication to provide different experiment characteristics for each run. Minor changes in the design of the earlier sections of the schema can allow for multiple instances of each component. The individual entries in each run-tuple will be selections from pools of each component type. The consistency checking that one would like among multiple runs with different components is not supported.

The Schema Language (Express)

The Express language is a minor component in the overall IIE design. Its features support the above specified schema contents, as well as a simple implementation. Designing a formal language for software systems experiment design constitutes a dissertation, not a thesis component. The schema design will certainly change, due to tool use and further SoftLab tool design; hence the language will also change. This provides further motivation for expending a minimum of effort on the design of Express.

The keyword approach makes for very simple parsing in the implementation. It also supports the ease of extension that is useful in a prototype design and implementation. One flaw in the design lies in its not supporting comments.

SchemaPrep

The `SchemaPrep` program reads the schema file and prepares for the construction of each of the experiment components. The idea for a separate program to prepare for component construction comes from Dr. Satyanarayanan's hardware simulator construction tool [16]. The program processes each section of the schema file separately. When an error occurs in one section of the schema the program may continue, depending on the nature of the error, to process the remaining schema sections, thereby isolating the parsing of as much of the schema as possible. Command line switches explicitly direct the processing of only selected sections. The user avoids the repeated parsing of correct sections by setting the correct switches.

The `SchemaPrep` program constructs hidden subdirectories under the current directory in the fashion of the Cambridge Modula-2 system [14]. Each subdirectory holds the files necessary for the preparation and construction of one run-time component. The user may ignore the contents of these directories. The sophisticated experimenter may make modifications in the directories contents, although this invalidates most of the consistency checking the IIE tools currently provide. We felt that the user should have easy access to the component files, during early use of this tool, for error checking and modification suggestions.

The execution of `SchemaPrep` causes the files `makefile` and `SchemaPrep.log` to appear in the current directory. The `makefile` supports the actual component construction stage of the experiment. The `SchemaPrep.log` file contains a log of the `SchemaPrep` program execution. This log provides useful documentation and supports a batch oriented execution. The contents of the sub-directories follow in the next sections.

The .exp_mgr Directory

The exp_mgr program carries out monitoring directives. The file `sensor` in the .exp_mgr subdirectory contains the enabled sensor list. A change in the design will allow multiple lists in the one file or multiple sensor list files with appropriate names. The file `main.c` contains the schema run-tuple information and permits an easy implementation of the multiple runs per-experiment feature. The file `makefile` supports the UNIX make utility used in the next stage of component construction.

The .hwconfig Directory

After running SchemaPrep this directory contains a copy of the selected hierarchy description file, a `.c` file related to the hierarchy description file, symbolic links to each of the device module files for the selected simulator, and a makefile to construct the simulator. The hierarchy descriptor file is a copy since device module substitutions require changes in its contents. The use of symbolic links for the device module files supports efficient use of the file system. The makefile and the `.c` file are the result of a component in the hardware simulator tool set. Their creation is not explicit in the design.

The .osconfig Directory

Symbolic links associate all of the components of a particular operating system with the `.osconfig` directory, with the exception of the main module. We use a copy of the main module to keep the design parallel with that for the simulator. The final specification for the contents of the main module was not known at the time of this design; it may permit, or perhaps require, useful modifications when making substitutions for the other modules. The parallel design of the component construction phase for both the simulator and operating system will aid in understanding and ease of implementation. A Modula-2 utility program requires the presence of the `m2path` file and constructs the `makefile`.

The .stimulus Directory

A copy of the user main module file resides here, as well as the supporting `m2path` file and `makefile`. A set of subdirectories under the `.stimulus` directory could easily support multiple workloads.

5.3 Experiment Component Construction

A successful execution of the program SchemaPrep places all of the necessary files, or links to them, in the appropriate sub-directory for each tool component. In addition, a `makefile` is in place in the experiment directory and in each sub-directory. The invocation of the main `makefile` in the experiment directory with the UNIX **make** utility program will result in the invocation of the `makefiles` in each of the sub-directories.

The top-level `makefile` provides the main control facility for component construction. Executing **make** with no arguments causes make to execute in each of the sub-directories. A file `Make.log` collects the output of the **make** programs and provides documentation on the component construction process. This file also supports batch oriented component construction. The user can invoke the **make** program and put it in the background to execute. At some later point, she can examine the contents of the `Make.log` file to ascertain which components were successfully constructed and which components contain errors that prevent their construction.

The execution of the make utility in a sub-directory will cause the re-construction of the associated component only if a change occurs in a depending file. This conditional execution is a feature of the **make** utility and its associated `makefile`. There is no need to provide any additional facility to support efficient component construction. Although the casual user should not tamper with the `makefiles`, these files do provide additional control to the sophisticated user to make modifications to experiment components that are not supported in the current implementation.

The **exp_mgr** is the only component that does not reside in its sub-directory. The top level `makefile` causes it to move to the experiment directory since its invocation actually executes an experiment. The other components are all manipulated by this program.

5.4 Experiment Execution

The experiment execution facility was the most difficult to design. The ideal facility would provide control over all parts of the simulation and all simulation components. To implement this requires an extensively instrumented hardware simulator, operating system, stimulus, and monitor. The design of such components and the means of their manipulation is far beyond the scope of this thesis. The experiment execution facility was therefore restricted to have no control at all during an individual run of an experiment. This decision allowed the current SoftLab hardware simulator to be used with only minor modifications and permitted an execution to take place without requiring inter-process communication.

After this decision, it was necessary to decide what kind of inter-run control should be provided. Section 5.2.8 presents the main reasons for the chosen approach.

Exp_mgr

The **exp_mgr** program executes an experiment. It first creates symbolic links in the experiment directory to the stimulus and operating system M-code files. These links can change between each run if the corresponding run-tuples in the schema file contain different operating system or stimulus selections. These links, along with the other configuration information contained in the current associated run-tuple, provide all of the information that is necessary for the start of the current simulation. The simulation now runs to completion. The file **sens.out** contains the record of the sensor invocations. The contents of this file are now interpreted by the **exp_mgr** program to produce the output for the current experiment run. The output is written to a file named in the associated run-tuple for the current run. The use of the file **sens.out** provides great flexibility for the implementation of the data analysis and presentation phase of the experiment.

5.5 The Primary SoftLab Components

The current SoftLab operating system and hardware simulator required minor modification to permit their integration in the IIE. For the most part, these modifications were driven by the selected embedded sensor design.

The Operating System

The SoftLab family of operating systems was designed with the SoftLab hardware simulator tool in mind. The design of the other run-time components, e.g., the monitor, was not complete at the time the operating systems were designed. This leads to minor modifications to the two current operating systems, Unibatch and Multibatch. Sensors, short Modula-2 code fragments, installed in the source code for the operating systems, will support the appropriate collection of data. Using code fragments provides easy modification or addition of sensors to support particular experiments. A standard set of sensors, with unique identifiers, for the operating systems has not been determined at this point.

The Hardware Simulator

The hardware simulator contained an internal monitor mechanism when received from Dr. Satyanarayanan. The modification of this mechanism should drive the design of the sensor facility in the simulator. The current design, for reasons of time, ignored this mechanism and relied on the addition of C code fragments for sensors. This provides the same ease of modification and addition as in the operating systems.

5.6 Additional Component Design

The Workload

The design of a full workload facility encompassing a family of operating systems and related hardware simulators is beyond the scope of this thesis. The properties of each of the run-time components, as well as the current SoftLab M-code interpreters in the hardware simulators, would have to be carefully studied to ensure that particular M-code stimulus routines had the desired characteristics. The current design concentrated on the stimulus selection mechanism rather than on the content of the stimulus routines. For this reason, the design specified Modula-2 routines as the building blocks for the stimulus libraries and supported the compilation of a Modula-2 workload. The main components of the schema interface to the stimulus facility were designed to support a smooth transition to M-code library routines. The user will still provide the control skeleton for the workload and select library routines with particular characteristics.

The Monitor

During the design of the monitor we encountered many of the same issues as we did during the design of the experiment manager. The monitor will collect information from experiment components implemented in different languages. Ideally, different kinds of data, i.e., data from different sensors, should be accessible at different times during the same experiment run. A valuable feature of a monitor in this experiment environment would support selection of data from particular sensors based on the current data being received. These issues led to design difficulties that could not be easily overcome within the constraints of the current thesis. The current design permits the selection of different sensors on a per-run basis. A design extension involving extensions to the schema and the monitor should be possible to allow selection of different sensors during the course of the same run. An extension to support sensor selection based on current data does not seem possible with the present approach. An overall design that supported a multiple process design with inter-process communication would be necessary to efficiently provide this extension. Section 6.6 in Chapter 6 looks at the implementation issues that affected the design of this component.

5.7 Design Evaluation

The design supports a workable implementation and provides an environment in which real experiments may be carried out, thereby meeting the primary goals set forth in the associated thesis proposal. A more ambitious design that addresses many of the problems pointed out in this chapter was originally intended. However, the necessity of a more focused design soon became apparent. Therefore, the overall mechanism of the IIE became the primary focus of the thesis.

Overall the design meets the requirements set out in Chapter 2. An experimenter has the tools at hand to

- specify an experiment: SchemaPrep; Express Language.
- construct an executable simulator: O/S family; Hardware Simulator; Workload Libraries.
- perform and evaluate an experiment: Monitor; exp_mgr.

In addition we used the UNIX makefile utility and options to `SchemaPrep` to easily modify and rerun existing experiments.

Design Strengths

The modular structure of the IIE components is the greatest strength, since it supports component extension and enhancement. Progress in extending or enhancing the design would be very slow if minor changes led to modifications in each of the components. We expect the design to grow and change as new SoftLab tools become available. Experiments that require multi-processor simulators and process oriented operating systems are a natural outgrowth of the modular design.

A second strength of the design also follows from the modularity. The modular design supports easy debugging and quick turn around time when synthesizing a particular simulator. If the user had to remake each of the IIE components each time she encountered errors in a component or component interface, very little experimentation could take place. For the student, the task would be to get the simulator running. The evaluation of the experiment and the lessons it might hold would become secondary.

Design Weaknesses

The inability of the experiment manager to influence the run-time components during a run is the major weakness in the design. The ability to effect changes in the simulator as a result of data received by the monitor is desirable. Experiments related to run-time tuning of component function are then possible. Experiment variations on a per-run basis can be too coarse for complex experiments involving many parameters. We feel that this weakness will become more problematic as the components grow in sophistication.

Another weakness that contributes to the problems with the exp_mgr mentioned above stems from allowing major IIE components to be specified in different high-level languages. On a superficial level portability is clearly an issue. On a more fundamental level interprocess communication is very difficult when the processes involved are written in different languages. An inter-process communication facility of some type will surely be necessary to provide intra-run modification supports.

6. The Implementation

The design of the IIE focuses on the experimental specification and execution process rather than on the function of the particular components of the experiment. Restricting the focus in this way allows for the completion of the design within the framework of this thesis. A similar restriction applies to this first implementation of the IIE. In this chapter, we present the major implementation decisions, as well as implementation details for each of the experiment components. Code fragments are often appropriate as part of the detailed exposition of the implementation. Those fragments included in subsequent sections of this chapter do not contain all of the comments and error handling statements that are in the actual implementation. We will evaluate the current implementation in the last section of this chapter.

6.1 Major Implementation Considerations

The completion of the SoftLab M-code compiler, the *UniBatch* operating system, and the *Bmachine* will produce the first fully working implementation of the SoftLab IIE. The first implementation of the SoftLab IIE contains sufficient function to be useful to both researchers and teachers. Extensive use by these two groups during the IIE development will lead to early detection of design flaws and critical implementation requirements. We felt that it was more important to implement a prototype IIE than to refine the IIE design, or extend its domain. A partial implementation is presently operational and was the basis of the experiment example in Appendix B of the IIE User Manual presented in Chapter 4 of this paper.

The UNIX program design philosophy calls for making use of existing tools to design new ones. In ths spirit, we incorporated numerous UNIX utilities, e.g., sed and make, into small C programs for quick implementation of the IIE. In addition to speed, this approach provides a very flexible framework for future modification. Future SoftLab designers and implementers can use the present implementation as both the model for a production level implementation and also as a base for

IIE design modifications and extension evaluation. The modular character of the implementation supports quick and easy incorporation of different implementations of particular functions and/or components.

The potential use of the IIE by whole classes of students requires that it work efficiently in both space and time. Symbolic links, a feature of the UNIX BSD4.2 file system, provide a simple mechanism to share files and avoid unnecessary copying. The use of these links, along with minimal use of intermediate support files, keeps disk resource use to a minimum. Design decisions regarding the splitting of the experiment process into preparation, integration and execution phases, as well as the selective nature of the SchemaPrep and make programs, lead to a time efficient implementation. The three primary support tools may all be run in the background, and hence may be scheduled to run during nighttime hours. Once an experiment begins execution, the overhead of the exp_mgr program is negligible in comparison to that of the operating system or hardware simulator.

The judicious use of symbolic links helps prevent the occurrence of potential security and inconsistency problems. Access to files via symbolic links falls under the same access permission restrictions as the original file instance. The possibilities of accidently modifying a file required by others or picking up the wrong version of a file are reduced substantially.

This implementation is flexible, easily modifiable, quite efficient in use of resources, and reasonably secure. These criteria were applied to select the proper path to take at each phase in the implementation process.

6.2 Experiment Specification

The program SchemaPrep reads the schema file a string at a time, where a string is any sequence of characters bounded by white space (see Appendix A of the SoftLab IIE User Manual for a description of white space). Express language keywords provide the structure necessary for parsing the input.

The keystruct structure contains: the bounding keywords for each main section in the schema, a pointer to the function responsible for processing that section, the associated command line switch, and a descriptive string for messages. An array of keystruct structures, keylist, is defined in the header file schemaprep.h.

```
struct keystruct {
      char *keyword;        /* Section Start Keyword */
      char *delimit;        /* Section End Keyword */
      int (*keyfunc)();     /* Associated Procedure */
      char sw_char;         /* Associated Command Line Switch */
      char *desc;           /* Descriptive String */
} keylist[] = {
      "BeginSchema", NULL, BeginProc, NULL, NULL,
      "InitSection", "EndInit", InitProc, 'i', "initial",
      "StimulusSection", "EndStimulus", StimProc, 'w', "stimulus",
      "HWConfigSection", "EndHW", HWCProc, 'h', "hardware",
      "OSConfigSection", "EndOS", OSCProc, 'o', "operating system",
      "SensorSection", "EndSensor", SensProc, 'm', "sensor",
      "DirectivesSection", "EndDirectives", DirProc, 'd', "directives",
      "EndSchema", NULL, EndProc, NULL, NULL
};
```

The entry for the hardware section of the schema contains the beginning keyword *HWConfigSection* and the ending keyword *EndHw*. The procedure HWCProc processes the body of the hardware configuration section.

```
HWCProc()
{

      /* Create the hidden directory for the hardware component */
      mkdir(".hwconfig",0755);
      fprintf(stdout,"In HWCProc \n");
      fprintf(lfp,"In HWCProc \n");

      /* Process the body of the section */
      hwcprep(sfp,lfp);
}
```

Actions that are part of the section processing but do not rely on the content of the section are performed first. The creation of the hidden directory is the only such action for the hardware component. The procedure hwcprep(), which processes the body of the section, has as parameters pointers to the schema file, sfp, and the log file, lfp. The procedures for the other sections of the schema are similar to HWCProc and hwcprep.

The UNIX system routine supplies the actions required for the main body processing for each schema section. The following code fragment is part of hwcprep() from the file hwcprep.c.

```
strcpy(command,"filesubs ");
strcat(command,buffer);
strcat(command," ");
strcat(command,buffer3);
strcat(command," ");
strcat(command,".hwconfig");
strcat(command,"/");
strcat(command,hdfile);
system(command);
```

Invoking `system()` executes the program `filesubs` that modifies the hierarchy description file in the hidden directory `.hwconfig`. The modification consists of substituting one device module name for another as specified in the schema. The program `filesubs` is a shell script that makes use of another UNIX utility, `sed`, to actually make the substitution. The body of `filesubs` follows.

```
# Substitute string $2 for string $1 in file $3.
sed -e "/$1/s?$1?$2?" $3 >!  temp
cp temp $3
/bin/rm temp
```

The procedures, programs, and code fragments listed above illustrate the rapid prototyping approach taken in implementing the `SchemaPrep` tool, based on existing UNIX utilities.

6.3 Experiment Component Construction

Along with the related makefiles, the UNIX `make` utility program supports the implementation for the component construction phase of the IIE. The schema directory level `makefile` actually invokes each component `makefile` via a shell script. The scripts redirect the output of the component `make` invocations to the file `Make.log` in the schema directory. The contents of the main `makefile` and the shell script that invoke `make` in the hardware hidden directory follow.

Main makefile

```
all:
        touch Make.log       # Create the log file
        makehw.sh            # Invoke make in .hwconfig
        makeos.sh            # Invoke make in .osconfig
        makestim.sh          # Invoke make in .stimulus
        makeexpmgr.sh        # Invoke make in .exp_mgr
clean:
        /bin/rm -f -r .exp_mgr
        /bin/rm -f -r .hwconfig
        /bin/rm -f -r .osconfig
        /bin/rm -f -r .stimulus
        /bin/rm -f Make.log
        /bin/rm -f OS.mcd Stimulus.mcd
        /bin/rm -f sens.out sim.log
        /bin/rm -f exp_mgr
```

Shell Script makehw.sh

```
cd .hwconfig; make "HOME = /usr/softlab" >>& ../Make.log
```

6.4 Experiment Execution

Main Program Module

The experiment manager directives section of the schema contains part of the entry procedure call to the supporting library routines. The `run-tuples` in the `DirectiveSection` section of the schema are the actual parameter lists to a procedure. The `SchemaPrep` program concatenates the string *DoRun* with each `run-tuple`, and writes the resulting string to the file `main.c` in the experiment manager hidden directory `.exp_mgr`. The following is an example of the contents of the `main.c` file following an execution of `SchemaPrep`.

```
main()
{
DoRun("Bmachine","UniBatch.out","main.out","sensors","out1");
DoRun("Bmachine","UniBatch.out","main.out","sensors","out2");
}
```

Running `make` in the `.exp_mgr` directory causes the compiling of the file `main.c` and the linking with the experiment manager library to produce the executable file `.exp_mgr`. The `makefile` below contains all that is necessary to produce `.exp_mgr` in the schema directory.

```
LIB = /unc/drm/sl/lib/libexp_mgr.a

install:  main.o $(LIB)
       cc main.o -o exp_mgr $(LIB)
       mv exp_mgr ..
```

Library Routines

The library entry routine, DoRun, contains three main sections, accomplishing simulation preparation, simulator invocation, and simulation output processing.

- The preparation section links the files containing the M-code for the operating system and the stimulus to files in the schema directory. Symbolic links to files in the schema directory hide the hidden directory structure from the simulator so that changes in the directory structure will not cause modifications in the simulator.

```
strcpy(buffer, ".osconfig/");
strcat(buffer, os);
symlink(buffer, "OS.mcd");
```

```
strcpy(buffer, ".stimulus/");
strcat(buffer, stim);
symlink(buffer, "Stimulus.mcd");
```

The main memory device module in the simulator expects to find `OS.mcd` and `Stimulus.mcd` in the current directory. The schema directory must be the current directory when the simulator executes.

- The experiment manager forks off a process to do the simulation. The experiment manager then waits until the process is complete before attempting to process the output.

```
pid = fork();

if (pid == 0) {

        /* in child */
        execl(simbuf, simbuf, "-d", "100", 0);
}
else {
        if (wait(0) != pid)
        else {
              ProcessOutput(out, sensbuf);
        }
}
```

The system utility `execl` overlays the current text segment in the newly forked process with the executable code in the process indicated by the contents of `simbuf`. The string in `simbuf` contains the full path to the executable hardware simulator in the hidden directory `.hwconfig`.

- The procedure `ProcessOutput` reads in the sensor information and prints out a processed sensor report. The active sensor list is read from the `sensor` file in the `.exp_mgr` directory.

```
for (i=0; i<MAX_SENSORS; i++)
        arr[i].id = -1;

while (fscanf(sfp, "%d", &newid) != EOF) {

        if (newid >= MAX_SENSORS) {
              continue;
        }

        arr[newid].id = newid;
        arr[newid].first = 0.0;
        arr[newid].last = 0.0;
        arr[newid].avg = 0.0;
        arr[newid].count = 0;

} /* end while */
```

Each `arr` array entry contains a structure for the information collected from one sensor. The `id` field of the structure distinguishes between sensors selected in the active sensor file, pointed to by `sfp`, and those not selected, i.e., those with an `id` set to negative one.

```
while (fscanf(ofp, "%d,%f", &inst, &stime) != EOF) {
```

```
if (inst >= MAX_SENSORS) {
        continue;
}

if (arr[inst].count == 0)
        arr[inst].first = stime;
arr[inst].count += 1;
arr[inst].last = stime;
arr[inst].avg = (stime - arr[inst].first) / arr[inst].count;

} /* end while */
```

The sensor output of the simulation is in the file pointed to by `ofp`. Each valid sensor record, i.e., the value of the sensor identifier is less than `MAX_SENSORS`, causes an update to the appropriate sensor structure. The active sensor list has no bearing on the processing of sensor information in the sensor output file. The output report code prints the sensor information only for those structures with an `id` field not equal to negative one. Processing all sensor information supports different implementations where information from the sensors loads directly into a database. The resultant queries to collect output report information would result from the contents of the active sensor list.

6.5 Primary SoftLab Components

The Hardware Simulator

The hardware section of the schema specifies a base machine selected from the SoftLab machine collection, and possible device module substitutions. The associated implementation task is to build an executable simulator from the proper components. Creating symbolic links in the hidden directory `.hwconfig` to the correct device module and hierarchy description files provides all of the components. The CPU module must contain the proper interpreter for the specified workload and operating system. As the user is responsible for specifying the correct components in the schema, the implementation contains no consistency checking mechanism. The main memory modules must contain the correct memory configuration and also the correct paths to the workload and operating system M-code files. Section 6.4 presents the solution to the main memory and M-code file consistency problem. Section 6.6 discusses the current handling of sensors in the hardware simulator.

The Operating Systems

There is currently no facility to insert sensors into the operating system. We felt that such a decision could better be made with a Modula-2 to M-code compiler in hand. To be amenable to the IIE implementation, the operating system does not require any additional modification. As mentioned in section 6.6, the user is responsible for specifying an operating system that is compatible with the other selected run-time components.

6.6 Additional Tool Implementation

Workload Library

The workload library must consist of M-code routines in order to behave in accordance with performance requirements. A library of Modula-2 routines would have performance characteristics that are dependent on the current Modula-2 to M-code compiler. It is probable that desired workload characteristics would not be attainable given a particular compiler. To generate a full library in M-code that contains routines covering the range of necessary performance characteristics would comprise a Master's Thesis in itself. The current implementation contains Modula-2 routines simply to demonstrate that the mechanism works. An additional file contains M-code instructions to exercise the sensor handling function of the M-code interpreter in the CPU module.

The Monitor Facility

The current implementation prints time-stamped information to a file for each sensor that the simulation reaches. The print statements are built into the simulator code. An M-code instruction will be modified and used for the sensor entries in the workload and in the compiled operating system. The SoftLab operating systems will require the incorporation of the necessary sensors, and the Modula-2 compiler must also change so as to recognize a Modula-2 sensor instruction.

6.7 Implementation Evaluation

Implementation Strengths

The flexibility of the implementation is its main strength. The implementer may easily substitute shell scripts, C code, or UNIX utilities for existing components. A flexible implementation is very important since we expect the implementation to change as new SoftLab tools come into being and the IIE takes on a wider range of uses. The modular structure of all the IIE components supports the ease of modification. The rapid prototyping approach that we took also led to a less rigid design since it resulted in a large number of external procedure calls.

The current implementation also supports easy extension. As more and more users experiment with the system, future implementers can promote more segments of the implementation to well-tuned C code. The overhead associated with calling the **system** utility and invoking shell scripts will then disappear.

Implementation Weaknesses

The component tool modification mechanisms are not sufficiently general. Various mechanisms are necessary for each component, and keeping modifications consistent is not easy. For example, sensors having the same identifier may be added to different components. No mechanism is in place that can check the current sensor set for conflicts. This in not a major weakness but may cause unnecessary confusion.

There is also no consistency checking mechanism to ensure that the workload, operating system, simulator, and sensors actually correspond to each other. This particular weakness will become more important as users start running experiments that entail a large number of different runs with differing components.

The current SoftLab Modula-2 to M-code compiler is not yet in place. A full test of the implementation is not possible without a working compiler, so that an operating system and workload can actually run on the simulator. The first full test may bring to light further inadequacies in the implementation that are currently hidden.

7. Conclusion and Future Work

We set out to design and implement a set of tools that comprises an Integrated Instrumentation Environment. We have successfully completed this task. Chapter 5 presents the design of the IIE and Chapter 6 presents the implementation. Chapter 4 illustrates the possible uses of the IIE in the sections describing the contents of the schema file. The various run-time systems, together with the possible monitor and experiment manager directives, that may be selected in a schema support a wide range of software and hardware configuration experiments. We developed EXPRESS as the experiment schema specification language to assist the user in experiment design.

The prototype of the IIE meets the functional requirements set forth in Chapter 2, as evidenced in the sample experiment run in Appendix B. Even though the M-code for the operating system and the stimulus were not available, a simple substitution of another M-code file allowed the sample experiment to complete. The ease of the substitution itself illustrates the flexibility of the implementation. Besides being useful in its own right the current IIE is a valuable tool for evaluation of future IIE's for SoftLab. The modular and flexible IIE permits modifications to design or implementation elements to test new ideas. The final sections of chapters 5 and 6 evaluate the implementation in more detail.

7.1 Future Work

We designed this IIE with operating systems experiments as the primary focus. However, we had other software system experiments in mind — most notably those related to compiler design. A valuable extension would tie the code generation phase of a compiler to the specification of the M-code interpreter on the target simulator. A new section in the schema containing interpreter directives, together with a modified SchemaPrep program, would then effect the appropriate modifications to the interpreter in the CPU device module of the selected hardware simulator.

Two functions currently relegated to the **exp_mgr** program are to process and present experiment data to the user. A new, more powerful program, taking over these functions, would take data and processing directives from the **exp_mgr** and provide sophisticated processing and result presentation. The schema would contain processing directives in the sensor directives section. No other component of the IIE would need to be changed.

Selected operating systems, simulators, and workloads must correspond if they are to work correctly together. The IIE user would like to know that the current set of sensors is consistent. The addition of consistency checking to the implementation could prevent many potential problems from occuring. We expect additional consistency problems to arise as the IIE grows to comprise more components.

A multi-process implementation of the **exp_mgr** program with inter-process communication support would provide access to the run-time components during an experiment run. A wide range of design decisions related to experiment control would arise if such an implementation existed.

The above modifications would extend the utility and the function of the IIE. However, none of these modifications are necessary to make the current IIE useful or efficient. The SoftLab IIE is a powerful and usable tool in its current form for both educational and research applications.

References

1. Ben-Ari, M. *Principles of Concurrent Programming.* Prentice-Hall, Inc., Englewood Cliffs, N.J. (1982).

2. Comer, D. *Operating System Design, the Xinu Approach.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1984).

3. Corbin, K., Corwin, W., Goodman, R., Hyde, E., Kramer, K., Werme, E, Wulf, W. **A Software Laboratory, Preliminary Report.** 71-104, Computer Science Department, Carnegie-Mellon University (August 1971).

4. Corwin, W., Wulf, W. **SL230–A Software Laboratory, Intermediate Report.** Computer Science Department, Carnegie-Mellon University (May 1972).

5. Dolotta, T.A., Haight, R.C., Mashey, J.R. Unix Time-Sharing System: The Programmer's Workbench. *Bell System Technical Journal 57*, 6 (1978), 2177-2200.

6. Feiler, P.H., Medina-Mora R. An Incremental Programming Environment. *IEEE Transactions on Software Engineering 7.SE-7*, 5 (Sept. 1981), 472.

7. Fisher, F. A. **A Family of Operating Systems in a Software Laboratory.** Master's Thesis, Computer Science Department, University of North Carolina at Chapel Hill (April 1986).

8. Goodwin, J.W. Why Programming Environments Need Dynamic Data Types. *IEEE Transactions on Software Engineering 7.SE-7*, 5 (Sept. 1981), 451.

9. Gutknecht, J. Tutorial on Modula-2. *Byte* (August 1984), 157-176.

10. Halstead, M.H. *A Laboratory Manual for Compiler and Operating System Implementation.* American Elsevier, New York, NY (1974).

11. Heindel, L.E., Roberto, J.J. *LANG-PAK–An Interactive Language Design System*, vol. 13. American Elsevier Pub. Co., New York, N.Y. (1975).

12. Holt, R.C., Graham, G.S., Lazowska, E.D., Scott, M.A. *Structured Concurrent Programming with Operating System Applications.* Addison-Wesley Pub. Co., Reading, M.A. (1978).

13. Kernighan, B.W., Ritchie, D.M. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N.J. (1978).

14. Cambridge University Computer Laboratory**Cambridge Modula-2 Manual Pages.** Cambridge University, Cambridge, England (1985).

15. Ohran, R. Lilith and Modula-2. *Byte 9,* 8 (August 1984), 181-192.

16. Satyanarayanan, M. **A Modularized Architectural Simulator.** (SoftLab Internal Working Document No. 2), Computer Science Department, University of North Carolina at Chapel Hill (April 1985).

17. Segall, Z., Singh, A., Snodgrass, R., Jones, A. J., Siewiorek, D. P. An Integrated Instrumentation Environment for Multiprocessors. *IEEE Transactions on Computers C-32* (January 1983), 4-14.

18. Shannon, K. **idlc Users Manual, (Version 2.0).** (SoftLab Document No. 8), Computer Science Department, University of North Carolina at Chapel Hill (December 1985).

19. Shaw, A. *Operating Systems* (The Series in Automatic Computation). Prentice-Hall, Inc., Englewood Cliffs, N.J. (1974).

20. Stroustrup, B. **A Set of C Classes for Co-Routine Style Programming.** Computing Science Technical Report No. 90, Bell Laboratories (November 1980).

21. Stroustrup, B. **Classes: An Abstract Data Type Facility for the C Language.** Computing Science Technical Report No. 84, Bell Laboratories (August 1981).

22. Wirth, N. History and Goals of Modula-2. *Byte* (August 1984), 145-152.

Appendix A

Express Language Syntax

This appendix provides the syntax for the Express language, a description of valid file contents for required files and valid file names. Terminals are in bold and italics. The terminals in italics are defined informally at the end of the listing, e.g., *slstimlib*. Those in bold are reserved keywords in the language, e.g., **BeginSchema**. The remaining strings are non-terminals, e.g., stimlib.

schemafile	::=	**BeginSchema** initsect stimsect hwsect ossect sensect dirsect **EndSchema**
initsect	::=	**InitSection EndInit**
stimsect	::=	**StimulusSection** stimlib stimfile **EndStimulus**
hwsect	::=	**HWConfigSection** hw hwhd hwdmsubs hwmodsubs **EndHW**
ossect	::=	**OSConfigSection** os osmain osdefsubs osimpsubs **EndOS**
sensect	::=	**SensorSection** senslist **EndSensor**
dirsect	::=	**DirectivesSection** runlist **EndDirectives**
stimlib	::=	**Library:** *slstimlib*
stimfile	::=	**File:** *userfile*
hw	::=	**Machine:** *slmachine*
hwhd	::=	**HD:** *slmachinehd*
hwdmsubs	::=	**DMSubs:** dmsub* **EndSubs**
os	::=	**OS:** *slos*
osmain	::=	**Main:** *slosmain*
osdefsubs	::=	**DefSubs:** osdefsub* **EndDefSubs**
osimpsubs	::=	**ImpSubs:** osimpsub* **EndImpSubs**
senslist	::=	**SensorList** sensentry* **EndSensorList**
runlist	::=	**RunList** runentry* **EndRunList**
dmsub	::=	*devicemodule userfile*

osdefsub	::=	*definitionfile userfile*
osimpsub	::=	*impfile userfile*
sensentry	::=	slist | *sfile*
runentry	::=	(*"slmachine"*,*"slosmain"*,*"sensin"*,*"userfile"*)
slist	::=	*number**

Terminals:

slmachine	The name of one of the SoftLab simulators.
slmachinehd	The name of a hierarchy description file for the selected simulator.
devicemodule	The name of a device module for the selected simulator.
slos	The name of a SoftLab operating systems.
slosmain	The name of a main module for the selected operating system.
definitionfile	The name of a definition module for the selected operating system.
impfile	The name of an implementation module for the selected operating system.
sfile	The name of a file containing an *slist.*
slstimlib	The name of one of the SoftLab stimulus libraries.
userfile	The name of a user supplied file.
number	A number is an unsigned integer.

White space consists of blanks, tabs and newlines. The name of a file can be an arbitrary path in the file system and is not to be quoted. Relative path names are relative to the directory containing the schema file. Names, not specifically file names, are identifiers of SoftLab components.

Appendix B

The Design and Execution of a Sample Experiment

The following example experiment contains two separate runs that differ only in the names of the output files that they produce. In this typescript, the following steps are performed:

- the current contents of the schema file directory are listed
- the contents of file `senslist` is printed
- the contents of the schema file, `example.sch`, is printed
- the program `SchemaPrep` is executed
- the new contents of the schema file directory are listed
- `make` is executed
- the new contents of the schema file directory are listed
- the `exp_mgr` is executed
- the final contents of the directory are listed
- the contents of the two experiment output files are listed

In this script, text typed by the user is bold italics.

```
% ls -a
senslist          subcpu.dm         submem.dm         example.sch
% cat senslist
12
24
% cat example.sch
BeginSchema

InitSection
        EndInit

StimulusSection
        Library:  lowref
        File:  /unc3/unc/drm/sl/src/Stimulus/main.mod
        EndStimulus

HWConfigSection
        Machine:  Bmachine
        HD: Bmachine.hd
        DMSubs:
                Bmainmem.dm submem.dm
                Bcpuint.dm subcpu.dm
        EndSubs
        EndHW

OSConfigSection
        OS: unibatch
        Main:  UniBatch.mod
        DefSubs:
                Loader.def /usr/softlab/src/OS/unibatch/Loader.def
        EndDefSubs
        ImpSubs:
                Loader.mod /usr/softlab/src/OS/unibatch/Loader.mod
        EndImpSubs
        EndOS

SensorSection
        SensorList:     1 5 8
                        -f senslist
                        45
        EndSensorList
        EndSensor

DirectivesSection
        RunList: ("Bmachine","UniBatch.out","main.out","sensors","out1")
                 ("Bmachine","UniBatch.out","main.out","sensors","out2")
        EndRunList
        EndDirectives

EndSchema
```

% SchemaPrep example.sch

```
In BeginProc
In InitProc
In StimProc
        Using Stimulus Library "/unc3/unc/drm/sl/src/Stimulus/lowref"
        Using Stimulus File "/unc3/unc/drm/sl/src/Stimulus/main.mod"
No match.
In HWCProc
        Using Machine "/unc3/unc/drm/sl/src/machine/Bmachine"
        Using Hierarchy Description File "Bmachine.hd"
        Substituting Device Module "submem.dm"
                for Device Module "Bmainmem.dm"
        Substituting Device Module "subcpu.dm"
                for Device Module "Bcpuint.dm"
In OSCProc
        Using Operating System "/unc3/unc/drm/sl/src/OS/unibatch"
        Using Main Module File "/unc3/unc/drm/sl/src/OS/unibatch/UniBatch.mod"
        Substituting Definition Module "/usr/softlab/src/OS/unibatch/Loader.def"
                for Definition Module "Loader.def"
        Substituting Implementation Module
                        "/usr/softlab/src/OS/unibatch/Loader.mod"
                for Implementation Module "Loader.mod"
In SensProc
```

```
        Using Sensor "1"
        Using Sensor "5"
        Using Sensor "8"
        Using Sensor File "senslist"
        Using Sensor "12"
        Using Sensor "24"
        Using Sensor "45"
In DirProc
Adding run tuple ("Bmachine","UniBatch.out","main.out","sensors","out1")
Adding run tuple ("Bmachine","UniBatch.out","main.out","sensors","out2")
In EndProc
% ls -a
total 50
.exp_mgr          .hwconfig         .osconfig         .stimulus
SchemaPrep.log    example.sch       makefile          senslist
subcpu.dm         submem.dm
% make
Making hardware simulator.
Making operating system.
Making stimulus.
Making experiment manager.
Experiment components successfully created.
Executable file is "exp_mgr".
% ls -a
.exp_mgr          .hwconfig         .osconfig         .stimulus
Make.log          SchemaPrep.log    example.sch       exp_mgr
makefile          senslist          subcpu.dm         submem.dm
% exp_mgr
Begining experiment
Begining experiment run number 1
Linking "OS.mcd" to ".osconfig/Unibatch.out"
Linking "Stimulus.mcd" to ".stimulus/main.out"
Invoking simulator ".hwconfig/Bmachine.sim"
Reading from enabled sensor file ".exp_mgr/sensors"
Reading from simulation sensor file "sens.out"
Processing sensor statistics
Writing output file "out1"
Experiment run number 1 completed
Begining experiment run number 2
Linking "OS.mcd" to ".osconfig/Unibatch.out"
Linking "Stimulus.mcd" to ".stimulus/main.out"
Invoking simulator ".hwconfig/Bmachine.sim"
Reading from enabled sensor file ".exp_mgr/sensors"
Reading from simulation sensor file "sens.out"
Writing output file "out2"
Experiment run number 2 completed
Ending experiment
% ls -a
.exp_mgr          .hwconfig         .osconfig         .stimulus
Make.log          OS.mcd            SchemaPrep.Log    Stimulus.mcd
example.sch       exp_mgr           makefile          out1
out2              sens.out          senslist          subcpu.dm
```

submem.dm

```
% cat out1
Sensor   First      Last     Average    Count
=============================================
     1  340.00    340.00        -           1
     5   40.00    320.00      93.33         4
     8    0.00      0.00        -           0
    12  100.00    260.00      32.00         6
    24  160.00    160.00        -           1
    45   20.00    360.00      48.57         8
% cat out2
Sensor   First      Last     Average    Count
=============================================
     1  340.00    340.00        -           1
     5   40.00    320.00      93.33         4
     8    0.00      0.00        -           0
    12  100.00    260.00      32.00         6
    24  160.00    160.00        -           1
    45   20.00    360.00      48.57         8
```

Appendix C

Manual Pages

NAME

SchemaPrep — IIE component preparation program

SYNOPSIS

SchemaPrep [-dhimow] schemafile

DESCRIPTION

A schema file describes an experiment by specifying each of the major components of the IIE run-time system and also by issuing directives that the experiment manager will take note of during experiment execution. The program *SchemaPrep* takes this schema file as input and prepares the experiment components for execution. A subdirectory for each component is created in the current directory along with a log of the component preparation process in the file SchemaPrep.log in the current directory. In addition, the file *Makefile* is created in the current directory. This file contains the UNIX *make* utility directives to construct the executable experiment manager program *exp_mgr.*

In addition to naming the schema file the user may also set command line switches. Setting a switch causes *SchemaPrep* to process the associated section of the schema. If no switch is set the program processes all sections of the schema. The switches have the following meanings:

-d	Interpret the directives section
-h	Interpret the hardware configuration section
-i	Interpret the initialization section
-m	Interpret the sensor section (for monitoring)
-o	Interpret the operating system section
-w	Interpret the stimulus section (the workload)

EXAMPLES

```
SchemaPrep schemafile
     #processes the entire schema in file schemafile and prepares all of the components.
SchemaPrep -dm schemafile
     #processes the directives and sensor sections of the schema in file schemafile
     #and prepares only the experiment manager component.
```

FILES

```
./.exp_mgr      /* experiment manager subdirectory */
./.stimulus     /* workload subdirectory */
./.hwconfig     /* hardware simulator subdirectory */
./.osconfig     /* operating system subdirectory */
./makefile      /* experiment construction makefile */
```

SEE ALSO

exp_mgr(1), make(1)
IIE Users Manual, by R. Morrill

RESTRICTIONS

The current working directory must contain the schema file. The log file is always appended and must be explicitly removed if its current contents are no longer useful.

ERRORS

The error messages are intended to be self explanatory.

AUTHOR

Richard Morrill
University of North Carolina at Chapel Hill

NAME

exp_mgr — IIE experiment execution program

SYNOPSIS

exp_mgr

DESCRIPTION

The user executes an IIE experiment by executing the program **exp_mgr** in the schema file home directory. The experiment will then run until all of the requested runs complete. The results of the experiment are written to the files specified in the run-tuples of the associated schema file. In addition to the results file, **exp_mgr** produces a file containing a log of the experiment in the same directory. This file is named `exp_mgr.log`. The log file contains information on the experiment specifications, the time the experiment was run and any warning or error messages produced as a result of the execution.

FILES

./exp_mgr.log /* experiment manager log */

SEE ALSO

SchemaPrep(1), make(1)
IIE Users Manual, by R. Morrill

RESTRICTIONS

The logging mechanism is not implemented.

ERRORS

The error messages are intended to be self explanatory.

AUTHOR

Richard Morrill
University of North Carolina at Chapel Hill

```
struct keystruct {
        char *keyword;
        char *delimit;
        int (*keyfunc)();
        char sw_char;
        char *desc;
} keylist[] = {
         "BeginSchema", NULL, BeginProc, NULL, NULL,
         "InitSection", "EndInit", InitProc, 'i', "initial",
         "StimulusSection", "EndStimulus", StimProc, 'w', "stimulus",
         "HWConfigSection", "EndHW", HWCProc, 'h', "hardware",
         "OSConfigSection", "EndOS", OSCProc, 'o', "operating system",
         "SensorSection", "EndSensor", SensProc, 'm', "sensor",
         "DirectivesSection", "EndDirectives", DirProc, 'd', "directives",
         "EndSchema", NULL, EndProc, NULL, NULL
};

#define MAXSWITCH 6
#define LOGFILE "SchemaPrep.log"
```

```
/*
 * FILE: SchemaPrep.c
 *
 * CONTENTS:
 *      main            Driver for the IIE schema preparation program.
 *      *Proc           Section processing routines.
 *      SwitchOn        Section switch on detector.
 *      InvalidSwitch   Invalid switch detector.
 *      SwitchError     Switch error help routine.
 *      Ignore          Section ignore, by consuming, routine.
 *
 * HISTORY:
 *
 */

#include <stdio.h>
#include "schema.h"             /* Schema program definitions and constants */
#include "schemaprep.h"         /* Driver definitions and constants */

static FILE *sfp;                       /* Schema file pointer */
static FILE *lfp;                       /* Schema log file pointer */
static char switches[MAXSWITCH + 1];    /* Switch array */
static char rcsid[] = "$$";             /* RCS ident string */

/*
 * PURPOSE:
 *      Driver for the IIE schema preparation program.
 *
 * RETURN CODES:
 *      None.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      None.
 */
main(argc,argv)
int argc;
char *argv[];
{
    char temp[MAXARGLEN+1];             /* Input buffer */
    char schemafile[MaxFileName + 1];   /* File name buffer */
    boolean fileflag;                   /* File name read flag */
    boolean switchflag;                 /* Switch read flag */
    int i;                              /* Loop index */
    char switchtemp[LINELENGTH + 1];    /* Switches read buffer */
    char buffer[MAXKEYLEN + 1];         /* Section keyword buffer */


    /*
     * Initialize.
     */
    fileflag = FALSE;
    switchflag = FALSE;

    /*
     * Command line parsing.
     */
    for (i=1; i<argc; i++) {

        sscanf(*++argv,"%s",temp);
        if (!strncmp(temp,"-",1)) {
            if (switchflag == TRUE) {
                fprintf(stderr,
                    "Warning: ignoring additional switches %s\n",temp);
            }
            else {

                sscanf(&temp[1],"%s",switchtemp);

                if (strlen(switchtemp) > MAXSWITCH) {
                    fprintf(stderr,"Error: too many switches \"%s\"\n\n",
                        switchtemp);
                    SwitchError();
                    exit(1);
                }

                else if (InvalidSwitch(switchtemp)) {
                    fprintf(stderr,"Error: invalid switch(es) \"%s\"\n\n",
                        switchtemp);
                    SwitchError();
                    exit(1);
                }

                strcpy(switches, switchtemp);
                switchflag = TRUE;
            }
        }
        else {
            if (fileflag == TRUE) {
                fprintf(stderr,"Warning: ignoring additional file %s\n",temp);
            }
            else {
                sscanf(temp,"%s",schemafile);
                fileflag = TRUE;
            }
        }

    } /* end for */

    /*
     * No schema file name in command line.
     */
    if (!fileflag) {
        fprintf(stderr,"Error: input file required\n");
        exit(1);
    }

    if (!(sfp = fopen(schemafile,"r"))) {
        fprintf(stderr,"Error: cannot open file %s\n",schemafile);
        exit(1);
    }
```

```
    if (!(lfp = fopen(LOGFILE,"a"))) {
        fprintf(stderr,"Error: cannot open or create file %s\n",
            LOGFILE);
        exit(1);
    }

    /*
     * No switches set so
     * select all schema sections.
     */
    if (!switchflag) {
        for (i=0; i<MAXSWITCH; i++) {
                switches[i] = keylist[i+1].sw_char;
        }
    }

    /*
     * Attempt to process all
     * eight sections.
     */
    for (i=0 ;i<8 ;i++) {
        fscanf(sfp,"%s",buffer);
        if (!(strcmp(buffer,keylist[i].keyword))) {

            /*
             * Invoke the section processing routine
             * for the current section.
             */
            if (SwitchOn(switches,keylist[i].sw_char))
                (*keylist[i].keyfunc)();

            /*
             * Consume the current section since
             * it was not selected for processing.
             */
            else {
                fprintf(stdout,"Ignoring \"%s\" section\n", keylist[i].desc);
                fprintf(lfp,"Ignoring \"%s\" section\n", keylist[i].desc);
                Ignore(keylist[i].delimit);
            }
        }
        else {
            fprintf(stdout,"no match: %s, keyword %s\n",buffer,
                            keylist[i].keyword);
            fprintf(lfp,"no match: %s, keyword %s\n",buffer,
                            keylist[i].keyword);
        }
    }

    close(sfp);
    close(lfp);

}

BeginProc()
{
    fprintf(stdout,"In BeginProc \n");
    fprintf(lfp,"In BeginProc \n");
}

InitProc()
{
    mkdir(".exp_mgr",0755);
    fprintf(stdout,"In InitProc \n");
    fprintf(lfp,"In InitProc \n");
    initprep(sfp,lfp);
}


StimProc()
{
    mkdir(".stimulus",0755);
    fprintf(stdout,"In StimProc \n");
    fprintf(lfp,"In StimProc \n");
    stimprep(sfp,lfp);
}


HWCProc()
{
    mkdir(".hwconfig",0755);
    fprintf(stdout,"In HWCProc \n");
    fprintf(lfp,"In HWCProc \n");
    hwcprep(sfp,lfp);
}


OSCProc()
{
    mkdir(".osconfig",0755);
    fprintf(stdout,"In OSCProc \n");
    fprintf(lfp,"In OSCProc \n");
    oscprep(sfp,lfp);
}


SensProc()
{
    fprintf(stdout,"In SensProc \n");
    fprintf(lfp,"In SensProc \n");
    sensprep(sfp,lfp);
}


DirProc()
{
    fprintf(stdout,"In DirProc \n");
    fprintf(lfp,"In DirProc \n");
    dirprep(sfp,lfp);
}

/*
```

```
 * Used for testing purposes.
 * Replace any processing function entry
 * in schemaprep.h with this call.
 */
DummyProc()
{
    fprintf(stdout,"In DummyProc \n");
    fprintf(lfp,"In DummyProc \n");
}

EndProc()
{
    fprintf(stdout,"In EndProc \n");
    fprintf(lfp,"In EndProc \n");
}

/*
 * PURPOSE:
 *      Consume the contents of the schema file
 *      from the current position up to and including
 *      the string 'string'.
 *
 * RETURN CODES:
 *      None.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      None.
 */
Ignore(string)
char *string;   /* IN --- section closing keywork */
{
    char buf[128+1];
    int eof_flag;

    eof_flag = TRUE;
    while (fscanf(sfp, "%s", buf) != EOF)
        if (!strcmp(buf,string)) {
            eof_flag = FALSE;
            break;
        }

    if (eof_flag)
        fprintf(stdout,"ERROR: Missing keyword \"%s\" \n", string);
        fprintf(lfp,"ERROR: Missing keyword \"%s\" \n", string);
}

/*
 * PURPOSE:
 *      Print the current switches to standard out
 *      and the log file.
 *
 * RETURN CODES:
 *      None.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      None.
 */
SwitchError()
{
    int i;              /* loop index */

    fprintf(stdout,"Valid switches are:\n");
    fprintf(lfp,"Valid switches are:\n");
    for (i=1; i<=MAXSWITCH; i++)
        fprintf(stdout,"\t\t\t\"%c\"---%s\n",
            keylist[i].sw_char, keylist[i].desc);
        fprintf(lfp,"\t\t\t\"%c\"---%s\n",
            keylist[i].sw_char, keylist[i].desc);
}

/*
 * PURPOSE:
 *      Check 's' to see if it contains only valid switches.
 *
 * RETURN CODES:
 *      TRUE if 's' contains an invalid switch specification;
 *      FALSE otherwise.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      Assume the 's' is not longer that MAXSWITCH characters.
 */
boolean InvalidSwitch(s)
char *s;        /* IN --- switch array */
{
    int i;              /* Array index */
    int switchcount;    /* Valid switch in switch array count */
    boolean invalid;    /* Return code */

    switchcount = 0;

    /*
     * Increment 'switchcount' for every valid
     * switch specified in 's'.
     */
    for(i=1; i<=MAXSWITCH; i++) {
        if (index(s,keylist[i].sw_char)) {
            switchcount += 1;
        }
    }

    /*
     * If all the switches are valid the
     * value of switchcount should be equal
     * to the number of characters in 's'.
     */
    invalid = (switchcount != strlen(s));
    return(invalid);
}

/*
 * PURPOSE:
 *      Check to see if 'c' is NULL or in the character array 's'.
```

```
 *
 * RETURN CODES:
 *      TRUE if 'c' is NULL or in character array 's';
 *      FALSE otherwise.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      None.
 */
SwitchOn(s,c)
char *s;        /* IN --- switch array */
char c;         /* IN --- current section switch or NULL */
{
    int on;     /* Return code */

    if (c == NULL)
        on = TRUE;

    else
        on = (index(s,c) != 0);

    return(on);
}
```

```
/*
 * FILE: init.c
 *
 * CONTENTS: IIE schema initialization section preparation routine.
 *
 * HISTORY: Written by Richard Morrill for SoftLab, UNC-CH, 1/1/86.
 *
 */

#include <stdio.h>
#include "schema.h"

/*
 * PURPOSE:
 *      Process the initialization section of a schema.
 *
 * RETURN CODES:
 *      None.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      None.
 */
initprep(sfp,lfp)
FILE *sfp;              /* IN --- schema file */
FILE *lfp;              /* IN --- schema log file */
{
    char buffer[128+1];         /* String building buffer */
    char command[128+1];        /* System command building buffer */


    /*
     * Create a link to the experiment manager
     * make file in the .exp_mgr directory.
     */
    strcpy(command, "ln -s ");
    strcat(command, EMMAKEFILE);
    strcat(command, " .exp_mgr/makefile");
    system(command);

    /*
     * Create a link to the main experiment
     * make file in the schema directory.
     */
    strcpy(command, "ln -s ");
    strcat(command, MAINMAKEFILE);
    strcat(command, " makefile");
    system(command);

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"EndInit")) {
        fprintf(stdout,"Keyword \"EndInit\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,"Keyword \"EndInit\" expected --- read \"%s\"\n",
            buffer);
    }
}
```

```
#include <stdio.h>
#include "schema.h"

stimprep(sfp,lfp)
FILE *sfp;
FILE *lfp;
{
    char buffer[128+1];
    char stimlib[128+1];
    char libpath[128+1];
    char stimfile[128+1];
    char stimpath[128+1];

    char command[300];

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"Library:")) {
        fprintf(stdout,
            "Keyword \"Library:\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,
            "Keyword \"Library:\" expected --- read \"%s\"\n",
            buffer);
    }
    else {
        fscanf(sfp,"%s",stimlib);
        strcpy(libpath,SLstimpath);
        strcat(libpath,"/");
        strcat(libpath,stimlib);
        fprintf(stdout,"\tUsing Stimulus Library \"%s\"\n",libpath);
        fprintf(lfp,"\tUsing Stimulus Library \"%s\"\n",libpath);
    }

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"File:")) {
        fprintf(stdout,"Keyword \"File:\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,"Keyword \"File:\" expected --- read \"%s\"\n",
            buffer);
    }
    else {
        fscanf(sfp,"%s",stimfile);

        fprintf(stdout,"\tUsing Stimulus File \"%s\"\n",stimfile);
        fprintf(lfp,"\tUsing Stimulus File \"%s\"\n",stimfile);

        strcpy(command,"ln ");
        strcat(command,stimfile);
        strcat(command," .stimulus/main.mod");
        system(command);
    }

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"EndStimulus")) {
        fprintf(stdout,
            "Keyword \"EndStimulus\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,
            "Keyword \"EndStimulus\" expected --- read \"%s\"\n",
            buffer);
    }

    else {
        strcpy(command, "cd .stimulus; m2hdir");
        system(command);

        strcpy(command, "cp /unc3/unc/drm/sl/bin/Stimm2path .stimulus/m2path");
        system(command);

        strcpy(command, "filesubs STIMDIR ");
        strcat(command, SLstimpath);
        strcat(command, " .stimulus/m2path");
        system(command);

        strcpy(command, "Stimmake.sh main.mod");
        system(command);
    }
}
```

```
/*
 * FILE: hwcprep.c
 *
 * CONTENTS: IIE hardware section processing routine.
 *
 * HISTORY: Written by Richard Morrill for SoftLab, UNC-CH, 1/1/86.
 */

#include <stdio.h>
#include "schema.h"

/*
 * PURPOSE:
 *      Process the hardware configuration section of the schema file.
 *
 * RETURN CODES:
 *      None.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      None.
 */
hwcprep(sfp,lfp)
FILE *sfp;                  /* IN --- Schema file */
FILE *lfp;                  /* IN --- Schema log file */
{
    int len;                        /* hierarchy description file name length */
    char buffer[128+1];             /* string building buffer */
    char buffer2[128+1];            /* string building buffer */
    char buffer3[128+1];            /* string building buffer */
    char machdir[128+1];            /* selected machine subdirectory */
    char dirpath[128+1];            /* path to SoftLab machine directory */
    char hdfile[128+1];             /* hierarchy description file name */
    char hdpath[128+1];             /* path to the hierarchy description file */

    char command[300];              /* system command buffer */

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"Machine:")) {
        fprintf(stdout,"Keyword \"Machine:\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,"Keyword \"Machine:\" expected --- read \"%s\"\n",
            buffer);
    }

    /*
     * Select machine.
     */
    else {
        fscanf(sfp,"%s",machdir);
        strcpy(dirpath,SLmachpath);
        strcat(dirpath,"/");
        strcat(dirpath,machdir);
        fprintf(stdout,"\tUsing Machine \"%s\"\n",dirpath);
        fprintf(lfp,"\tUsing Machine \"%s\"\n",dirpath);
    }
    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"HD:")) {
        fprintf(stdout,"Keyword \"HD:\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,"Keyword \"HD:\" expected --- read \"%s\"\n",
            buffer);
    }

    /*
     * Create a link in the .hwconfig directory
     * to the selected hierarchy description file.
     */
    else {
        fscanf(sfp,"%s",hdfile);
        fprintf(stdout,"\tUsing Hierarchy Description File \"%s\"\n",hdfile);
        fprintf(lfp,"\tUsing Hierarchy Description File \"%s\"\n",hdfile);

        strcpy(hdpath,dirpath);
        strcat(hdpath,"/");
        strcat(hdpath,hdfile);
        strcpy(command,"ln -s ");
        strcat(command,hdpath);
        strcat(command," .hwconfig");
        system(command);
    }

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"DMSubs:")) {
        fprintf(stdout,"Keyword \"DMSubs:\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,"Keyword \"DMSubs:\" expected --- read \"%s\"\n",
            buffer);
    }

    /*
     * Make the specified substitutions in the simulator
     * by linking to alternate device module files.
     */
    else {
        while (fscanf(sfp,"%s",buffer) != EOF) {

            if (!strcmp(buffer,"EndSubs"))
                break;

            fscanf(sfp,"%s",buffer2);
            fprintf(stdout,"\tSubstituting Device Module \"%s\"\n",buffer2);
            fprintf(lfp,"\tSubstituting Device Module \"%s\"\n",buffer2);
            fprintf(stdout,"\t          for Device Module \"%s\"\n",buffer);
            fprintf(lfp,"\t         for Device Module \"%s\"\n",buffer);

            if (strncmp(buffer2,"/",1)) {
                strcpy(buffer3,"../");
                strcat(buffer3,buffer2);
            }
            else
                strcpy(buffer3,buffer2);
```

```
            strcpy(command,"ln -s ");
            strcat(command,buffer3);
            strcat(command," .hwconfig/");
            strcat(command,buffer);
            system(command);

 /* Take care of relative path names
    in the hierarchy description file
    device module replacement implementation.

            if (strncmp(buffer2,"/",1)) {
                strcpy(buffer3,"../");
                strcat(buffer3,buffer2);
            }
            else
                strcpy(buffer3,buffer2);

            strcpy(command,"filesubs ");
            strcat(command,buffer);
            strcat(command," ");
            strcat(command,buffer3);
            strcat(command," ");
            strcat(command,".hwconfig");
            strcat(command,"/");
            strcat(command,hdfile);
            system(command);                */
        }
    }

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"EndHW")) {
        fprintf(stdout,"Keyword \"EndHW\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,"Keyword \"EndHW\" expected --- read \"%s\"\n",
            buffer);
    }

    else {

        /*
         * link in the rest of the .dm files
         */
        strcpy(command, "chdir .hwconfig; LnDMFile.sh ");
        strcat(command,dirpath);
        system(command);

        /*
         * strip ".hd" suffix from hierarchy description file
         */
        len = strlen(hdfile);
        strncpy(buffer, hdfile, len-3);
        strcpy(&buffer[len-3], "");

        /*
```

```
         * Execute the command to prepare
         * the hierarchy description file
         * and create the make file for the
         * selected device modules.
         */
        strcpy(command,"cd .hwconfig;");
        strcat(command,"hdprep ");
        strcat(command,buffer);
        system(command);
    }
}
```

```
/*
 * FILE: oscprep.c
 *
 * CONTENTS: IIE operating system schema section preparation routine.
 *
 * HISTORY: Written by Richard Morrill for SoftLab, UNC-CH, 1/1/86.
 */

#include <stdio.h>
#include "schema.h"

/*
 * PURPOSE:
 *
 * RETURN CODES:
 *
 * RESTRICTIONS/ASSUMPTIONS:
 */
oscprep(sfp,lfp)
FILE *sfp;                  /* IN --- schema file */
FILE *lfp;                  /* IN --- schema log file */
{
    char buffer[128+1];         /* string building buffer */
    char buffer2[128+1];        /* string building buffer */
    char osdir[128+1];          /* selected operating system subdirectory */
    char dirpath[128+1];        /* SoftLab operating system directory */
    char mainfile[128+1];       /* main module file for selected O/S */
    char mainpath[128+1];       /* main module file path */

    char command[300];          /* system command buffer */

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"OS:")) {
        fprintf(stdout,"Keyword \"OS:\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,"Keyword \"OS:\" expected --- read \"%s\"\n",
            buffer);
    }

    /*
     * Select operating system.
     */
    else {
        fscanf(sfp,"%s",osdir);
        strcpy(dirpath,SLospath);
        strcat(dirpath,"/");
        strcat(dirpath,osdir);
        fprintf(stdout,"\tUsing Operating System \"%s\"\n",dirpath);
        fprintf(lfp,"\tUsing Operating System \"%s\"\n",dirpath);
    }

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"Main:")) {
        fprintf(stdout,"Keyword \"Main:\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,"Keyword \"Main:\" expected --- read \"%s\"\n",
            buffer);
    }

    /*
     * Select operating system main module.
     */
    else {
        fscanf(sfp,"%s",mainfile);
        strcpy(mainpath,dirpath);
        strcat(mainpath,"/");
        strcat(mainpath,mainfile);
        fprintf(stdout,"\tUsing Main Module File \"%s\"\n",mainpath);
        fprintf(lfp,"\tUsing Main Module File \"%s\"\n",mainpath);
    }

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"DefSubs:")) {
        fprintf(stdout,"Keyword \"DefSubs:\" expected -- read \"%s\"\n",
            buffer);
        fprintf(lfp,"Keyword \"DefSubs:\" expected -- read \"%s\"\n",
            buffer);
    }

    /*
     * Make the appropriate definition module substitutions
     * by creating links to alternate files in the .osconfig
     * subdirectory.
     */
    else {
        while (fscanf(sfp,"%s",buffer) != EOF) {
            if (!strcmp(buffer,"EndDefSubs")) {
                break;
            }
            fscanf(sfp,"%s",buffer2);
            fprintf(stdout,
                "\tSubstituting Definition Module \"%s\"\n",
                buffer2);
            fprintf(lfp,"\tSubstituting Definition Module \"%s\"\n",
                buffer2);
            fprintf(stdout,
                "\t          for Definition Module \"%s\"\n",
                buffer);
            fprintf(lfp,"\t          for Definition Module \"%s\"\n",
                buffer);
            strcpy(command,"ln -s ");
            strcat(command,buffer2);
            strcat(command," .osconfig/");
            strcat(command,buffer);
            system(command);
        }
    }

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"ImpSubs:")) {
        fprintf(stdout,"Keyword \"ImpSubs:\" expected -- read \"%s\"\n",
            buffer);
```

```
        fprintf(lfp,"Keyword \"ImpSubs:\" expected -- read \"%s\"\n",
            buffer);
    }

    /*
     * Make the appropriate implementation module substitutions
     * by creating links to alternate files in the .osconfig
     * subdirectory.
     */
    else {
        while (fscanf(sfp,"%s",buffer) != EOF) {

            if (!strcmp(buffer,"EndImpSubs")) {
                break;
            }

            fscanf(sfp,"%s",buffer2);
            fprintf(stdout,
                "\tSubstituting Implementation Module \"%s\"\n",
                buffer2);
            fprintf(lfp,
                "\tSubstituting Implementation Module \"%s\"\n",
                buffer2);
            fprintf(stdout,
                "\t          for Implementation Module \"%s\"\n",
                buffer);
            fprintf(lfp,
                "\t          for Implementation Module \"%s\"\n",
                buffer);
            strcpy(command,"ln -s ");
            strcat(command,buffer2);
            strcat(command," .osconfig/");
            strcat(command,buffer);
            system(command);
        }
    }

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"EndOS")) {
        fprintf(stdout,"Keyword \"EndOS\" expected -- read \"%s\"\n",
            buffer);
        fprintf(lfp,"Keyword \"EndOS\" expected -- read \"%s\"\n",
            buffer);
    }

    else {

        /*
         * Make the hidden Modula-2 hidden directories
         * in the .osconfig subdirectory.
         */
        strcpy(command, "chdir .osconfig; m2hdir");
        system(command);

        /*
         * Create links to all definition and implementation
         * modules without current links in the .osconfig
         * subdirectory.   Only substituted modules should
         * currently have links.
         */
        strcpy(command, "chdir .osconfig; LnDirFile.sh ");
        strcat(command,dirpath);
        system(command);

        /*
         * Create the makefile for the current operating
         * system in the .osconfig subdirectory.
         */
        strcpy(command, "chdir .osconfig; /bin/rm makefile ");
        system(command);
        strcpy(command, "mm2m.sh ");
        strcat(command, mainfile);
        system(command);

    }

}
```

```
/*
 * FILE: sensprep.c
 *
 * CONTENTS: IIE sensor section preparation routine.
 *
 * HISTORY: Written by Richard Morrill for SoftLab, UNC-CH, 1/1/86.
 */

#include <stdio.h>
#include "schema.h"

/*
 * PURPOSE:
 *      Process the sensor section for the schema.
 *
 * RETURN CODES:
 *      None.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      None.
 */
sensprep(sfp,lfp)
FILE *sfp;              /* IN --- schema file */
FILE *lfp;              /* IN --- schema log file */
{
    char buffer[128+1];         /* string building buffer */
    char sensfile[128+1];       /* sensor file path buffer */

    char command[300];          /* system command buffer */

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"SensorList:")) {
        fprintf(stdout,
            "Keyword \"SensorList:\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,
            "Keyword \"SensorList:\" expected --- read \"%s\"\n",
            buffer);
    }
    else {

        /*
         * Create the sensor file
         * in the .exp_mgr subdirectory.
         */
        strcpy(command,"touch");
        strcat(command," .exp_mgr/sensors");
        system(command);

        /*
         * Build the enabled sensor list
         * until the end of the section is reached.
         */
        while (fscanf(sfp,"%s",sensfile) != EOF) {

            if (!strcmp(sensfile,"EndSensorList"))
```

```
                break;

            /*
             * Add the sensors from a file.
             */
            if (!strncmp(sensfile,"-",1))
            {
                fscanf(sfp,"%s",sensfile);
                fprintf(stdout,
                    "\tUsing Sensor File \"%s\"\n",sensfile);
                fprintf(lfp,
                    "\tUsing Sensor File \"%s\"\n",sensfile);
                strcpy(command,"cat ");
            }

            /*
             * Add individual sensors.
             */
            else
            {
                fprintf(stdout,
                    "\tUsing Sensor \"%s\"\n",sensfile);
                fprintf(lfp,
                    "\tUsing Sensor \"%s\"\n",sensfile);
                strcpy(command,"echo ");
            }

            strcat(command,sensfile);
            strcat(command," >> .exp_mgr/sensors");
            system(command);

        } /* end while */
    } /* end else */


    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"EndSensor")) {
        fprintf(stdout,
            "Keyword \"EndSensor\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,
            "Keyword \"EndSensor\" expected --- read \"%s\"\n",
            buffer);
    }

}
```

```
/*
 * FILE: dirprep.c
 *
 * CONTENTS: IIE Schema directive section processing routine.
 *
 * HISTORY: Written by Richard Morrill for SoftLab, UNC-CH, 1/1/86.
 */

#include <stdio.h>
#include "schema.h"

/*
 * PURPOSE:
 *      Process the directive section of a schema file.
 *
 * RETURN CODES:
 *      None.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      None.
 */
dirprep(sfp,lfp)
FILE *sfp;                 /* IN --- Schema file */
FILE *lfp;                 /* IN --- Schema log file */
{
    int initflag;                  /* Run-tuple added flag */
    char buffer[128+1];            /* String building buffer */
    char command[128+1];           /* System command buffer */


    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"RunList:")) {
        fprintf(stdout,"Keyword \"RunList:\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,"Keyword \"RunList:\" expected --- read \"%s\"\n",
            buffer);
    }


    else {

        initflag = 0;

        /*
         * Create a new main.c file in the
         * .exp_mgr subdirectory.
         */
        strcpy(command,"/bin/rm -f ");
        strcat(command, EXPMAIN);
        system(command);
        strcpy(command,"cp ");
        strcat(command, SLEXPMAIN);
        strcat(command," ");
        strcat(command, EXPMAIN);
        system(command);
        /*
         * Process to the end of the section
         * or the end of the file;
         * whichever comes first.
         */
        while (fscanf(sfp,"%s",buffer) != EOF) {

            if (!strcmp(buffer,"EndRunList"))
                break;

            else {
                if (!initflag)
                    initflag = 1;

                fprintf(stdout,"Adding run tuple %s\n", buffer);
                fprintf(lfp,"Adding run tuple %s\n", buffer);

                /*
                 * Append the current run-tuple
                 * to main.c in the .exp_mgr
                 * subdirectory.
                 */
                strcpy(command, "echo \'");
                strcat(command, RUN_PROCEDURE);
                strcat(command, buffer);
                strcat(command, ";\' >> ");
                strcat(command, EXPMAIN);
                system(command);
            }
        } /* end while */

        /*
         * Close off the main.c file in the
         * .exp_mgr subdirectory.
         */
        strcpy(command, "cat ");
        strcat(command, SLEXPMAINEND);
        strcat(command, " >> ");
        strcat(command, EXPMAIN);
        system(command);

        if (!initflag) {
            fprintf(stdout,
             "No run tuples where added; the simulator will not be invoked.\n");
            fprintf(lfp,
             "No run tuples where added; the simulator will not be invoked.\n");
        }

    } /* end else */

    fscanf(sfp,"%s",buffer);
    if (strcmp(buffer,"EndDirectives")) {
        fprintf(stdout,"Keyword \"EndDirectives\" expected --- read \"%s\"\n",
            buffer);
        fprintf(lfp,"Keyword \"EndDirectives\" expected --- read \"%s\"\n",
            buffer);
```

```
    }
}
```

```
#define TRUE 1
#define FALSE 0

#define MAXARGLEN 20
#define MaxFileName 20
#define MAXKEYLEN  20
#define LINELENGTH 80
#define PATHTAIL ":/usr/lib/local/modula2"
#define RUN_PROCEDURE "DoRun"
#define EXPMAIN ".exp_mgr/main.c"
#define SLEXPMAIN "/unc/drm/sl/src/exp_mgr/main.c"
#define SLEXPMAINEND "/unc/drm/sl/src/exp_mgr/mainend"
#define EMMAKEFILE "/unc/drm/sl/src/make/emMakefile"
#define MAINMAKEFILE "/unc/drm/sl/src/make/MainMakefile"

typedef int boolean;

int BeginProc();
int InitProc();
int StimProc();
int HWCProc();
int OSCProc();
int SensProc();
int DirProc();
int DummyProc();
int EndProc();

static char *SLmachpath = "/unc3/unc/drm/sl/src/machine";
static char *SLospath = "/unc3/unc/drm/sl/src/OS";
static char *SLstimpath = "/unc3/unc/drm/sl/src/Stimulus";
```

```
/*
 * FILE: output.c
 *
 * CONTENTS:
 *      The simulation output routine for the experiment manager
 *      library libexp.a.
 *
 * HISTORY: Written by Richard Morrill for SoftLab, UNC-CH, 1/1/86.
 *
 */

#include <stdio.h>
#include "exp_mgr.h"

/*
 * Sensor information.
 */
struct sensor_struct {
    int id;             /* Sensor id */
    float first;        /* First time the sensor was reached */
    float last;         /* Last time the sensor was reached */
    float avg;          /* Average duration between sensor invocations */
    int count;          /* Number of times the sensor was reached */
};

typedef struct sensor_struct sentype;

/*
 * PURPOSE:
 *      Calculate the appropriate sensor information from the simulator
 *      output file and write it to the selected results file, 'out'.
 *
 * RETURN CODES:
 *      None.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      None.
 */
ProcessOutput(out, sens)

char *out;              /* IN --- experiment output results file */
char *sens;             /* IN --- enabled sensors file */
{
    sentype arr[MAX_SENSORS];   /* sensor array */
    int i;                      /* loop index */
    int newid;                  /* new sensor id */
    int inst;                   /* new sensor instance from sim.log */
    float stime;                /* new sensor time form sim.log */
    FILE *ofp;                  /* simulator output results file */
    FILE *xfp;                  /* experiment output results file */
    FILE *sfp;                  /* enabled sensors file */

    sfp = fopen(sens, "r");
    if (sfp == (FILE *) NULL) {
        fprintf(stderr,
        "Unable to open sensor file %s, no output processed.\n", sens);
```

```
        return(-1);
    }

    ofp = fopen("sens.out", "r");
    if (ofp == (FILE *) NULL) {
        fprintf(stderr,
        "Unable to read simulator sensor file %s, no output processed.\n",
                                "sens.out");
        fclose(sfp);
        return(-1);
    }

    xfp = fopen(out, "w");
    if (xfp == (FILE *) NULL) {
        fprintf(stderr,
        "Unable to create results file %s, no output processed.\n",
                                    out);
        fclose(sfp);
        fclose(ofp);
        return(-1);
    }

    /*
     * Initialize sensor array.
     */
    for (i=0; i<MAX_SENSORS; i++)
        arr[i].id = -1;

    fprintf(stdout, "Reading from enabled sensor file \"%s\"\n",sens);

    /*
     * Distinguish enabled sensors.
     */
    while (fscanf(sfp, "%d", &newid) != EOF) {

        if (newid >= MAX_SENSORS) {
            fprintf(stdout,
                "Invalid sensor \"%d\" read from sensor file %s.\n",
                                newid, sens);
            continue;
        }

        arr[newid].id = newid;
        arr[newid].first = 0.0;
        arr[newid].last = 0.0;
        arr[newid].avg = 0.0;
        arr[newid].count = 0;

    } /* end while */

    fprintf(stdout,"Reading from simulation sensor file \"%s\"\n","sens.out");

    /*
     * Process sensor information.
     */
    while (fscanf(ofp, "%d,%f", &inst, &stime) != EOF) {
        if (inst >= MAX_SENSORS) {
        fprintf(stdout,
            "Invalid sensor \"%d\" read from simulator log file %s.\n",
                                inst, "sens.out");
        continue;
        }

        if (arr[inst].count == 0)
        arr[inst].first = stime;
        arr[inst].count += 1;
        arr[inst].last = stime;
        arr[inst].avg = (stime - arr[inst].first) / arr[inst].count;
    } /* end while */

    fprintf(stdout,"Writing output file \"%s\"\n",out);
    fprintf(xfp, "Sensor   First      Last    Average   Count\n");
    fprintf(xfp, "_________________________________________\n\n");

    /*
     * Write out enabled sensor
     * information.
     */
    for (i=0; i<MAX_SENSORS; i++) {
        if (arr[i].id != -1)
        fprintf(xfp,
         "  %2d    %6.2f    %6.2f    %6.2f        %d\n\n",
            i, arr[i].first, arr[i].last, arr[i].avg, arr[i].count);
    } /* end for */

    fclose(ofp);
    fclose(sfp);
    fclose(xfp);

}
```

```
#! /bin/csh -f

# Execute the shell script 'LnFile.sh' on all
# the directories under the current directory,
# with each directory, the first command line argument, and
# the current working directory as arguments.

set ourh = `/bin/pwd`

find . -type d -exec LnFile.sh {} $1 $ourh \;
```

```
#! /bin/csh -f

# Change to the directory selected by
# arguments 3 and 1, and create symbolic
# links to all files under 2/1.

chdir $3/$1
ln -s $2/$1/* .
```

```
#! /bin/csh -f

setenv M2PATH `\bin\pwd`:OSDIR:/usr/lib/local/modula2
```

```
#!/bin/csh -f

# Make the .osconfig directory the current
# working directory, set the M2PATH variable
# and create the Modula-2 make file.

chdir `/bin/pwd`/.osconfig
source m2path
mm2m $1
```

```
#! /bin/csh -f

setenv M2PATH .:STIMDIR:/usr/lib/local/modula2
```

```
#!/bin/csh -f

# Make the .stimulus directory the current
# working directory, set the M2PATH variable
# and create the Modula-2 make file.

chdir `/bin/pwd`/.stimulus
source m2path
mm2m $1
```

```
#! /bin/csh -f

# Find all occurences of the first argument
# in the file named by the third argument.
# In the selected file replace all occurrences
# of argument one with argument two.

sed -e "/$1/s?$1?$2?" $3 >! temp
cp temp $3
/bin/rm temp
```

```
#! /bin/csh -f

setenv M2PATH .:OSDIR:/usr/lib/local/modula2
```

```
#!/bin/csh -f

# Exucute make in the .exp_mgr subdirectory and
# append the output to Make.log in the current
# directory.

cd .exp_mgr; make >>& ../Make.log
```

```
#!/bin/csh -f

# Execute the make program in the .hwconfig subdirectory
# and append the output to file Make.log in the current
# directory.

cd .hwconfig; make "HOME = /usr/softlab" >>& ../Make.log
```

```
#!/bin/csh -f


# Execute make in the .osconfig subdirectory and append
# the output to the file Make.log in the current
# directory.

cd .osconfig; source m2path ; make >>& ../Make.log
```

```
#!/bin/csh -f

# Execute the make program in the .stimulus subdirectory
# and append the output to the file Make.log in the current
# working directory.  Set the M2PATH variable since this is
# a Modula-2 make.

cd .stimulus; source m2path ; make >>& ../Make.log
```

```
#!/bin/csh -f

# Set up the M2PATH variable in the .osconfig
# subdirectory and create the associated make file.

chdir `/bin/pwd`/.osconfig
source m2path
mm2m $1
```

```
/*
 * FILE: exp_mgr.h
 *
 * CONTENTS: Definitions and constants used by the libexp.a routines.
 *
 * HISTORY: Written by Richard Morrill for SoftLab, UNC-CH, 1/1/86.
 */

/* Maximum number of sensors allowed.
 */
#define MAX_SENSORS 100
```

```
/*
 * FILE: dorun.c
 *
 * CONTENTS: DoRun - Entry routine to the exp_mgr run-time library (libexp.a).
 *
 * HISTORY: Written by Richard Morrill for SoftLab, UNC-CH, 1/1/86
 *
 */

#include <stdio.h>

static int count = 1;    /* experiment number */

/*
 * PURPOSE:
 *      Executes one full run of an experiment including the output
 *      file processing.
 *
 * RETURN CODES:
 *      None.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      No validity checking is done for the experiment components.
 */
DoRun(sim, os, stim, sens, out)

char *sim;       /* IN - executable simulator file */
char *os;        /* IN - os mcode file */
char *stim;      /* IN - stimulus mcode file */
char *sens;      /* IN - enabled sensors file */
char *out;       /* IN - simulator output results file */
{

    int pid;                    /* process id returned from fork() */
    char sensbuf[128+1];        /* full path to the sensor file */
    char simbuf[128+1];         /* full path to the simulator */


    fprintf(stdout, "Beginning experiment run number %d\n",count);

    /* Link the stimulus and os files to the names
     * expected in the main memory module.
     */
    InitFiles(os,stim);

    /* Build relative paths in.
     */
    strcpy(simbuf, ".hwconfig/");
    strcat(simbuf, sim);
    strcat(simbuf, ".sim");
    strcpy(sensbuf, ".exp_mgr/");
    strcat(sensbuf, sens);

    /* printf("forking\n"); */

    pid = fork();
    /* printf("pid = %d\n",pid); */

    if (pid == 0) {

        /* In the child.
         */
        fprintf(stdout, "Invoking simulator \"%s%s%s\"\n", ".hwconfig/", sim,
            ".sim");
        execl(simbuf, simbuf, "-d", "100", 0);
    }
    else {

        /* In the parent.
         */
        if (wait(0) != pid)
            fprintf(stdout,
                "Error in simulator invocation\n");
        else {
            fprintf(stdout, "Experiment run number %d completed\n",count);
            count++;
            ProcessOutput(out, sensbuf);
        }
    }
}
```

```
/*
 * FILE: init.c
 *
 * CONTENTS: Initialization routine for the libexp.a routine library.
 *
 * HISTORY: Written by Richard Morrill for SoftLab, UNC-CH, 1/1/86.
 */

#include <stdio.h>

/*
 * PURPOSE:
 *      Set up links in the schema directory for the O/S and
 *      stimulus M-code files.
 *
 * RETURN CODES:
 *      None.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      Assumes that the simulator is looking for the files as named
 *      in the current working directory.
 */
InitFiles(os, stim)

char *os;               /* IN --- O/S M-code file */
char *stim;             /* IN --- stimulus M-code file */

{
    char buffer[128+1];

    fprintf(stdout,"Linking \"%s\" to \"%s%s\"\n","OS.mcd", ".osconfig/",
        os);

    strcpy(buffer, ".osconfig/");
    strcat(buffer, os);
    symlink(buffer, "OS.mcd");

    fprintf(stdout,"Linking \"%s\" to \"%s%s\"\n","Stimulus.mcd", ".stimulus/",
        stim);

    strcpy(buffer, ".stimulus/");
    strcat(buffer, stim);
    symlink(buffer, "Stimulus.mcd");
}
```

```
        if (inst >= MAX_SENSORS) {
        fprintf(stdout,
            "Invalid sensor \"%d\" read from simulator log file %s.\n",
                                inst, "sens.out");
        continue;
        }

        if (arr[inst].count == 0)
        arr[inst].first = stime;
        arr[inst].count += 1;
        arr[inst].last = stime;
        arr[inst].avg = (stime - arr[inst].first) / arr[inst].count;
    } /* end while */

    fprintf(stdout,"Writing output file \"%s\"\n",out);
    fprintf(xfp, "Sensor   First      Last    Average   Count\n");
    fprintf(xfp, "__________________________________________\n\n");

    /*
     * Write out enabled sensor
     * information.
     */
    for (i=0; i<MAX_SENSORS; i++) {
        if (arr[i].id != -1)
        fprintf(xfp,
         "  %2d    %6.2f    %6.2f    %6.2f        %d\n\n",
            i, arr[i].first, arr[i].last, arr[i].avg, arr[i].count);
    } /* end for */

    fclose(ofp);
    fclose(sfp);
    fclose(xfp);

}
```

```
#! /bin/csh -f

# Execute the shell script 'LnFile.sh' on all
# the directories under the current directory,
# with each directory, the first command line argument, and
# the current working directory as arguments.

set ourh = `/bin/pwd`

find . -type d -exec LnFile.sh {} $1 $ourh \;
```

```
#! /bin/csh -f

# Change to the directory selected by
# arguments 3 and 1, and create symbolic
# links to all files under 2/1.

chdir $3/$1
ln -s $2/$1/* .
```

```
#! /bin/csh -f

setenv M2PATH `\bin\pwd`:OSDIR:/usr/lib/local/modula2
```

```
#!/bin/csh -f

# Make the .osconfig directory the current
# working directory, set the M2PATH variable
# and create the Modula-2 make file.

chdir `/bin/pwd`/.osconfig
source m2path
mm2m $1
```

```
#! /bin/csh -f

setenv M2PATH .:STIMDIR:/usr/lib/local/modula2
```

```
#!/bin/csh -f

# Make the .stimulus directory the current
# working directory, set the M2PATH variable
# and create the Modula-2 make file.

chdir `/bin/pwd`/.stimulus
source m2path
mm2m $1
```

Apr 22 20:24 1986 filesubs Page 1

```
#! /bin/csh -f

# Find all occurences of the first argument
# in the file named by the third argument.
# In the selected file replace all occurrences
# of argument one with argument two.

sed -e "/$1/s?$1?$2?" $3 >! temp
cp temp $3
/bin/rm temp
```

Nov 6 20:17 1985 m2path Page 1

```
#! /bin/csh -f

setenv M2PATH .:OSDIR:/usr/lib/local/modula2
```

Apr 22 20:25 1986 makeexpmgr.sh Page 1

```
#!/bin/csh -f

# Exucute make in the .exp_mgr subdirectory and
# append the output to Make.log in the current
# directory.

cd .exp_mgr; make >>& ../Make.log
```

Apr 22 20:27 1986 makehw.sh Page 1

```
#!/bin/csh -f

# Execute the make program in the .hwconfig subdirectory
# and append the output to file Make.log in the current
# directory.

cd .hwconfig; make "HOME = /usr/softlab" >>& ../Make.log
```

```
#!/bin/csh -f


# Execute make in the .osconfig subdirectory and append
# the output to the file Make.log in the current
# directory.

cd .osconfig; source m2path ; make >>& ../Make.log
```

```
#!/bin/csh -f

# Execute the make program in the .stimulus subdirectory
# and append the output to the file Make.log in the current
# working directory.  Set the M2PATH variable since this is
# a Modula-2 make.

cd .stimulus; source m2path ; make >>& ../Make.log
```

```
#!/bin/csh -f

# Set up the M2PATH variable in the .osconfig
# subdirectory and create the associated make file.

chdir `/bin/pwd`/.osconfig
source m2path
mm2m $1
```

```
/*
 * FILE: exp_mgr.h
 *
 * CONTENTS: Definitions and constants used by the libexp.a routines.
 *
 * HISTORY: Written by Richard Morrill for SoftLab, UNC-CH, 1/1/86.
 */

/* Maximum number of sensors allowed.
 */
#define MAX_SENSORS 100
```

```
/*
 * FILE: dorun.c
 *
 * CONTENTS: DoRun - Entry routine to the exp_mgr run-time library (libexp.a).
 *
 * HISTORY: Written by Richard Morrill for SoftLab, UNC-CH, 1/1/86
 *
 */

#include <stdio.h>

static int count = 1;    /* experiment number */

/*
 * PURPOSE:
 *      Executes one full run of an experiment including the output
 *      file processing.
 *
 * RETURN CODES:
 *      None.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      No validity checking is done for the experiment components.
 */
DoRun(sim, os, stim, sens, out)

char *sim;      /* IN - executable simulator file */
char *os;       /* IN - os mcode file */
char *stim;     /* IN - stimulus mcode file */
char *sens;     /* IN - enabled sensors file */
char *out;      /* IN - simulator output results file */
{

    int pid;                    /* process id returned from fork() */
    char sensbuf[128+1];        /* full path to the sensor file */
    char simbuf[128+1];         /* full path to the simulator */


    fprintf(stdout, "Beginning experiment run number %d\n",count);

    /* Link the stimulus and os files to the names
     * expected in the main memory module.
     */
    InitFiles(os,stim);

    /* Build relative paths in.
     */
    strcpy(simbuf, ".hwconfig/");
    strcat(simbuf, sim);
    strcat(simbuf, ".sim");
    strcpy(sensbuf, ".exp_mgr/");
    strcat(sensbuf, sens);

    /* printf("forking\n"); */

    pid = fork();
    /* printf("pid = %d\n",pid); */

    if (pid == 0) {

        /* In the child.
         */
        fprintf(stdout, "Invoking simulator \"%s%s%s\"\n", ".hwconfig/", sim,
            ".sim");
        execl(simbuf, simbuf, "-d", "100", 0);
    }
    else {

        /* In the parent.
         */
        if (wait(0) != pid)
            fprintf(stdout,
                "Error in simulator invocation\n");
        else {
            fprintf(stdout, "Experiment run number %d completed\n",count);
            count++;
            ProcessOutput(out, sensbuf);
        }
    }
}
```

```
/*
 * FILE: init.c
 *
 * CONTENTS: Initialization routine for the libexp.a routine library.
 *
 * HISTORY: Written by Richard Morrill for SoftLab, UNC-CH, 1/1/86.
 */

#include <stdio.h>

/*
 * PURPOSE:
 *      Set up links in the schema directory for the O/S and
 *      stimulus M-code files.
 *
 * RETURN CODES:
 *      None.
 *
 * RESTRICTIONS/ASSUMPTIONS:
 *      Assumes that the simulator is looking for the files as named
 *      in the current working directory.
 */
InitFiles(os, stim)

char *os;                   /* IN --- O/S M-code file */
char *stim;                 /* IN --- stimulus M-code file */

{
    char buffer[128+1];

    fprintf(stdout,"Linking \"%s\" to \"%s%s\"\n","OS.mcd", ".osconfig/",
        os);

    strcpy(buffer, ".osconfig/");
    strcat(buffer, os);
    symlink(buffer, "OS.mcd");

    fprintf(stdout,"Linking \"%s\" to \"%s%s\"\n","Stimulus.mcd", ".stimulus/",
        stim);

    strcpy(buffer, ".stimulus/");
    strcat(buffer, stim);
    symlink(buffer, "Stimulus.mcd");
}
```